

Daniel Danciu

George Mardale

ARTA PROGRAMĂRII ÎN JAVA

vol I - Concepte fundamentale

Cuprins

1	Prezentarea limbajului Java	17
1.1	Limbaje de programare	17
1.2	Începuturile limbajului Java	19
1.3	Caracteristici ale limbajului Java	22
1.4	Implementări ale limbajului Java	23
1.5	Implementarea Sun a limbajului Java	26
1.5.1	Platformele Java	26
1.5.2	Platforma J2SE (Java 2 Platform, Standard Edition) . . .	27
1.5.3	J2SDK (Java 2 SDK, Standard Edition)	29
1.5.4	J2RE (Java 2 Runtime Environment, Standard Edition) .	32
1.5.5	Documentația J2SE (Java 2 Standard Edition Documen- tation)	32
1.5.6	Licența de utilizare a platformei J2SE	32
2	Noțiuni fundamentale de programare în Java	34
2.1	Mediul de lucru Java	35
2.2	Primul program Java	35
2.2.1	Comentarii	36
2.2.2	Funcția main	37
2.2.3	Afișarea pe ecran	37
2.3	Tipuri de date primitive	37
2.3.1	Tipurile primitive	37
2.3.2	Constante	38
2.3.3	Declararea și inițializarea tipurilor primitive în Java . . .	39
2.3.4	Citire/scriere la terminal	40
2.4	Operatori de bază	40
2.4.1	Operatori de atribuire	41
2.4.2	Operatori aritmetici binari	42

2.4.3	Operatori aritmetici unari	42
2.4.4	Conversii de tip	43
2.5	Instrucțiuni condiționale	44
2.5.1	Operatori relaționali	44
2.5.2	Operatori logici	44
2.5.3	Operatori la nivel de bit	46
2.5.4	Instrucțiunea if	49
2.5.5	Instrucțiunea while	51
2.5.6	Instrucțiunea for	51
2.5.7	Instrucțiunea do	52
2.5.8	Instrucțiunile break și continue	53
2.5.9	Instrucțiunea switch	54
2.5.10	Operatorul condițional	56
2.6	Metode	56
2.6.1	Supraîncărcarea numelor la metode	57
3	Referințe	64
3.1	Ce este o referință?	64
3.2	Fundamente despre obiecte și referințe	67
3.2.1	Operatorul punct (.)	67
3.2.2	Declararea obiectelor	68
3.2.3	Colectarea de gunoaie (garbage collection)	69
3.2.4	Semnificația operatorului =	69
3.2.5	Transmiterea de parametri	70
3.2.6	Semnificația operatorului ==	71
3.2.7	Supraîncărcarea operatorilor pentru obiecte	72
3.3	Șiruri de caractere (stringuri)	72
3.3.1	Fundamentele utilizării stringurilor	72
3.3.2	Concatenarea stringurilor	73
3.3.3	Compararea stringurilor	74
3.3.4	Alte metode pentru stringuri	75
3.3.5	Conversia de la string la tipurile primitive și invers	79
3.4	Șiruri	79
3.4.1	Declarație, atribuire și metode	80
3.4.2	Expansiunea dinamică a șirurilor	83
3.4.3	Șiruri cu mai multe dimensiuni	86
3.4.4	Argumente în linie de comandă	86

4	Obiecte și clase	91
4.1	Ce este programarea orientată pe obiecte?	92
4.2	Un exemplu simplu	94
4.3	Metode uzuale	96
4.3.1	Constructori	96
4.3.2	Modificatori și accesorii	98
4.3.3	Afișare și <code>toString()</code>	98
4.3.4	Metoda <code>equals()</code>	99
4.3.5	Metode statice	99
4.3.6	Atribute statice	99
4.3.7	Metoda <code>main()</code>	101
4.4	Pachete	101
4.4.1	Directiva <code>import</code>	102
4.4.2	Instrucțiunea <code>package</code>	103
4.4.3	Variabila sistem <code>CLASSPATH</code>	103
4.4.4	Reguli de vizibilitate <code>package-friendly</code>	105
4.4.5	Compilarea separată	106
4.5	Alte operații cu obiecte și clase	106
4.5.1	Referința <code>this</code>	106
4.5.2	Prescurtarea <code>this</code> pentru constructori	108
4.5.3	Operatorul <code>instanceof</code>	108
4.5.4	Inițializatori statici	108
5	Moștenire	115
5.1	Ce este moștenirea?	116
5.2	Sintaxa de bază Java	119
5.2.1	Reguli de vizibilitate	120
5.2.2	Constructor și <code>super</code>	120
5.2.3	Redefinirea metodelor. Polimorfism	122
5.2.4	Redefinirea parțială a metodelor	128
5.2.5	Metode și clase <code>final</code>	128
5.2.6	Metode și clase abstracte	130
5.3	Exemplu: Extinderea clasei <code>Shape</code>	133
5.4	Moștenire multiplă	138
5.5	Interfețe	139
5.5.1	Definirea unei interfețe	139
5.5.2	Implementarea unei interfețe	139
5.5.3	Interfețe multiple	141
5.6	Implementarea de componente generice	141
5.7	Clase interioare (inner classes)	145

5.7.1	Clasificarea claselor interioare	151
5.7.2	Clasele membre statice ale clasei exterioare	151
5.7.3	Clasele membre nestatice ale clasei exterioare	153
5.7.4	Clase locale	153
5.7.5	Clase anonime	156
5.7.6	Avantajele și dezavantajele claselor interioare	159
5.8	Identificarea tipurilor de date în faza de execuție	160
5.8.1	Identificarea standard a tipurilor de date	163
5.8.2	Mecanismul de reflecție ("reflection")	165
6	Tratarea excepțiilor	178
6.1	Ce sunt excepțiile?	178
6.2	Tipuri de excepții	180
6.3	Definirea propriilor excepții de către programator	183
6.4	Prinderea și tratarea excepțiilor	186
6.4.1	Blocul try	186
6.4.2	Blocul catch	187
6.4.3	Blocul finally	191
6.5	Aruncarea excepțiilor	195
6.5.1	Instrucțiunea throw	195
6.5.2	Clauza throws	197
6.6	Sugestii pentru utilizarea eficientă a excepțiilor	200
6.7	Avantajele utilizării excepțiilor. Concluzii	204
7	Intrare și ieșire (Java I/O)	207
7.1	Preliminarii	207
7.2	Operații de bază pe fluxuri (stream-uri)	208
7.3	Obiectul StringTokenizer	209
7.4	Fișiere secvențiale	210
7.5	Utilizarea colecțiilor de resurse (resource bundles)	212
7.5.1	Utilizarea fișierelor de proprietăți pentru citirea datelor	212
8	Fire de execuție	216
8.1	Ce este un fir de execuție?	217
8.2	Fire de execuție în Java	218
8.2.1	Crearea firelor de execuție	218
8.3	Accesul concurent la resurse	222
8.3.1	Sincronizare	229
8.3.2	Capcana metodelor nesincronizate	233
8.3.3	Instrucțiunea synchronized	235

8.3.4	Competiție asupra monitoarelor	237
8.3.5	Sincronizarea metodelor statice	237
8.3.6	Monitoare și attribute publice	239
8.3.7	Când NU trebuie să folosim sincronizarea	239
8.3.8	Blocare circulară (deadlock)	240
8.4	Coordonarea firelor de execuție	242
8.4.1	De ce este necesar să coordonăm firele de execuție? . . .	242
8.4.2	wait() și notify()	244
8.4.3	Exemplu de coordonare a firelor de execuție: problema consumator-producător	245
8.4.4	Deținerea monitoarelor	250
8.4.5	Utilizarea instrucțiunii synchronized	251
8.4.6	Un alt exemplu: consumatori și producători multipli . .	255
8.4.7	Utilizarea lui InterruptedException	258
A	Mediul Java și uneltele ajutătoare	265
A.1	Editarea codului sursă	265
A.1.1	Editoare	265
A.1.2	Medii integrate IDE (Integrated Development Environ- ments)	266
A.2	Instalarea mediului Java	267
A.2.1	Instalarea sub Windows	268
A.2.1.1	Instrucțiuni de instalare	268
A.2.2	Instalarea sub Linux	270
A.2.2.1	Instrucțiuni de instalare	270
A.2.2.2	Instalare cu autoextragere	271
A.2.2.3	Instalarea folosind RPM	272
A.3	Compilarea codului sursă	273
A.3.1	Ant	275
A.4	Rularea unei aplicații java	277
A.4.1	CLASSPATH	277
A.5	Documentații java	278
B	Convenții de notație în Java. Utilitarul javadoc	279
B.1	Necesitatea convențiilor de scriere a programelor	279
B.2	Tipuri de fișiere	280
B.3	Organizarea fișierelor sursă .java	280
B.3.1	Comentariile de început	281
B.3.2	Instrucțiunile package și import	281
B.3.3	Declarațiile clasei și interfeței	282

B.4	Indentarea	283
B.4.1	Lungimea unei linii de cod	284
B.4.2	"Ruperea" liniilor	284
B.4.3	Acoladele	284
B.4.4	Spațierea	285
B.5	Comentariile	286
B.5.1	Comentarii de implementare	287
B.5.2	Comentariile de documentație	288
B.6	Declarări	288
B.6.1	Numărul de declarări pe linie	288
B.6.2	Inițializarea variabilelor	289
B.6.3	Locul de declarare a variabilelor	289
B.6.4	Declararea claselor și a interfețelor	289
B.7	Instrucțiuni	290
B.7.1	Instrucțiuni simple	290
B.7.2	Instrucțiuni compuse	290
B.7.3	Instrucțiunea return	291
B.7.4	Instrucțiunea if-else	291
B.7.5	Instrucțiunea for	292
B.7.6	Instrucțiunea while	292
B.7.7	Instrucțiunea do-while	292
B.7.8	Instrucțiunea switch	292
B.7.9	Instrucțiunea try-catch	293
B.8	Convenții de notație	294
B.9	Practici utile în programare	296
B.9.1	Referirea atributelor și a metodelor statice	296
B.9.2	Atribuirea de valori variabilelor	297
B.10	Exemplu de fișier sursă Java	297
B.11	Utilitarul javadoc	299
B.11.1	Sintaxa javadoc	299
B.11.2	Descrierea utilitarului	300
B.11.3	Etichete javadoc	303
B.11.4	Opțiunile comenzii javadoc	305
B.11.5	Exemple simple de utilizare javadoc	307
C	Definirea pachetelor Java. Arhive jar	310
C.1	Prezentare generală a pachetelor Java predefinite	310
C.2	Definirea de pachete de către utilizator	314
C.3	Arhive jar (Java ARchives)	322
C.3.1	Utilizarea fișierelor .jar	323

C.3.2	Crearea unui fișier .jar	323
C.3.3	Vizualizarea conținutului unui fișier .jar	327
C.3.4	Extragerea conținutului unui fișier .jar	328
C.3.5	Fișierul manifest al unei arhive jar	329
C.3.6	Modificarea fișierului manifest al unei arhive jar . . .	331
C.3.7	Modificarea conținutului unui fișier .jar	332
C.3.8	Rularea aplicațiilor Java "împachetate" într-o arhivă jar	333
D Internaționalizarea aplicațiilor.		
	Colecții de resurse	335
D.1	Timpul, numerele și datele calendaristice	335
D.2	Colecțiile de resurse în internaționalizarea aplicațiilor	341
D.2.1	Un exemplu concret	342
D.2.2	Implementarea colecțiilor prin ListResourceBundle	344
D.2.3	Implementarea colecțiilor prin PropertyResourceBundle	346
D.2.4	Utilizarea colecțiilor de resurse	348
D.3	Internaționalizarea mesajelor dinamice	351
E Resurse Java		
		356
E.1	Site-ul Sun Microsystems	356
E.2	Tutoriale despre limbajul Java disponibile pe Internet	357
E.3	Cărți despre limbajul Java disponibile pe Internet	357
E.4	Reviste online de specialitate	358
E.5	Liste de discuții despre limbajul Java	358
E.6	Alte resurse Java	359

Introducere

“Ce învățătură predă Maestrul?” -
întrebă un vizitator.
“Nici una”, răspunse discipolul.
“Atunci de ce țiine discursuri?”
“El doar ne arată calea - nu ne
învață nimic.”

Anthony de Mello, O clipă de
înțelepciune

Oricine a folosit cel puțin o dată Internetul sau a citit o revistă de specialitate în domeniul informaticii, a auzit cu siguranță cuvântul „Java”. Java reprezintă un limbaj de programare, creat de compania Sun Microsystems în anul 1995. Inițial, Java a fost gândit pentru a îmbunătăți conținutul paginilor web prin adăugarea unui conținut dinamic: animație, multimedia etc. În momentul lansării sale, Java a revoluționat Internetul, deoarece era prima tehnologie care oferea un astfel de conținut. Ulterior au apărut și alte tehnologii asemănătoare (cum ar fi Microsoft ActiveX sau Macromedia Shockwave¹), dar Java și-a păstrat importanța deosebită pe care a dobândit-o în rândul programatorilor, în primul rând datorită facilităților pe care le oferea. Începând cu anul 1998, când a apărut versiunea 2 a limbajului (engl. Java 2 Platform), Java a fost extins, acoperind și alte direcții de dezvoltare: programarea aplicațiilor *enterprise* (aplicații de tip server), precum și a celor adresate dispozitivelor cu resurse limitate, cum ar fi telefoane mobile, *pager*-e sau PDA-uri². Toate acestea au reprezentat facilități

¹ Cei care utilizează mai des Internetul sunt probabil obișnuiți cu controale ActiveX sau cu animații flash în cadrul paginilor web.

² PDA = Personal Digital Assistant (mici dispozitive de calcul, de dimensiuni puțin mai mari decât ale unui telefon mobil, capabile să ofere facilități de agendă, dar și să ruleze aplicații într-un mod relativ asemănător cu cel al unui PC). La momentul actual există mai multe tipuri de PDA-uri: palm-uri, pocketPC-uri, etc.

noi adăugate limbajului, care a păstrat însă și posibilitățile de a crea aplicații standard, de tipul aplicațiilor în linie de comandă sau aplicații bazate pe GUI³. Lansarea versiunii 2 a limbajului Java a fost o dovadă a succesului imens de care s-au bucurat versiunile anterioare ale limbajului, dar și a dezvoltării limbajului în sine, a evoluției sale ascendente din punct de vedere al facilităților pe care le oferă, cât și al performanțelor pe care le realizează.

Cum este organizată această carte?

Având în vedere popularitatea extraordinară de care se bucură limbajul Java în cadrul programatorilor din întreaga lume, am considerat utilă scrierea unei lucrări în limba română care să fie accesibilă celor care doresc să învețe sau să aprofundeze acest limbaj. Ideea care a stat la baza realizării acestei cărți a fost aceea de a prezenta nu numai limbajul Java în sine, ci și modul în care se implementează algoritmi și structurile de date fundamentale în Java, elemente care sunt indispensabile oricărui programator cu pretenții. Așadar, cartea nu este destinată numai celor care doresc să acumuleze noțiuni despre limbajul Java în sine, ci și celor care intenționează să își aprofundeze și rafineze cunoștințele despre algoritmi și să își dezvolte un stil de programare elegant. Ca o consecință, am structurat cartea în două volume: prima volum (cel de față) este orientat spre prezentarea principalelor caracteristici ale limbajului Java, în timp ce volumul al doilea (disponibil separat) constituie o abordare a algoritmilor din perspectiva limbajului Java. Finalul primului volum cuprinde un grup de cinci anexe, care prezintă mai amănunțit anumite informații cu caracter mai special, deosebit de utile pentru cei care ajung să programeze în Java. Am ales această strategie deoarece a dobândi cunoștințe despre limbajul Java, fără a avea o imagine clară despre algoritmi, nu reprezintă un progres real pentru un programator. Iar scopul nostru este acela de a vă oferi posibilitatea să deveniți un programator pentru care limbajul Java și algoritmi să nu mai constituie o necunoscută.

Cele două volume cuprind numeroase soluții Java complete ale problemelor prezentate. Mai este necesară o remarcă: deseori am optat, atât în redactarea codului sursă cât și în prezentarea teoretică a limbajului sau a algoritmilor, pentru păstrarea terminologiei în limba engleză în defavoarea limbii române. Am luat această decizie, ținând cont de faptul că mulți termeni s-au consacrat în acest format, iar o eventuală traducere a lor le-ar fi denaturat înțelesul.

Primul volum cuprinde opt capitole:

Capitolul 1 reprezintă o prezentare de ansamblu a tehnologiei Java. Capitolul debutează cu istoria limbajului, începând cu prima versiune și până la cea

³GUI = Graphical User Interface, interfață grafică de comunicare cu utilizatorul, cum sunt în general aplicațiile disponibile pe sistemul de operare Microsoft Windows.

curentă. În continuare, sunt înfățișate câteva detalii despre implementările existente ale limbajului. Implementarea Java a firmei Sun este tratată separat, în detaliu, fiind și cea pe care s-au testat aplicațiile realizate pe parcursul cărții.

Capitolul 2 este cel care dă startul prezentării propriu-zise a limbajului Java, începând cu crearea și executarea unui program simplu. Sunt prezentate apoi tipurile primitive de date, constantele, declararea și inițializarea variabilelor, operatorii de bază, conversiile, instrucțiunile standard și metodele.

Capitolul 3 este destinat referințelor și obiectelor. Sunt prezentate în detaliu noțiunile de referință, obiect, șiruri de caractere (String-uri) și de șiruri de elemente cu dimensiuni variabile.

Capitolul 4 continuă prezentarea începută în capitolul anterior, prezentând modul în care se pot defini propriile clase în Java și cum se implementează conceptele fundamentale ale programării orientate pe obiecte.

Capitolul 5 prezintă în detaliu un principiu esențial al programării orientate pe obiecte: *moștenirea*. Sunt prezentate de asemenea noțiuni adiacente cum ar fi cea de polimorfism, interfață, clasă interioară, identificare a tipurilor de date în faza de execuție (RTTI = Runtime Type Identification).

Capitolul 6 este dedicat în întregime modului de tratare a excepțiilor în Java. Se prezintă tipurile de excepții existente în limbajul Java, cum se pot defini propriile tipuri de excepții, cum se pot prinde și trata excepțiile aruncate de o aplicație. Finalul capitolului este rezervat unei scurte liste de sugestii referitoare la utilizarea eficientă a excepțiilor.

Capitolul 7 prezintă sistemul de intrare-ieșire (I/O) oferit de limbajul Java. Pe lângă operațiile standard realizate pe fluxurile de date (*stream*) și fișiere secvențiale, este prezentată și noțiunea de *colecție de resurse* (engl. resource bundles).

Capitolul 8 este rezervat problemei delicate a firelor de execuție (*thread-uri*). Pornind cu informații simple despre firele de execuție, se continuă cu accesul concurent la resurse, sincronizare, monitoare, coordonarea firelor de execuție, cititorul dobândind în final o imagine completă asupra sistemului de lucru pe mai multe fire de execuție, așa cum este el atât de elegant realizat în Java.

Cele opt capitole ale primului volum sunt urmate de un grup de anexe, care conțin multe informații utile programatorilor Java.

Anexa A constituie o listă cu editoarele și mediile integrate pentru dezvoltarea aplicațiilor Java, precum și un mic tutorial de realizare și executare a unei aplicații Java simple. Tot aici este prezentată și *ant*, o unealtă de foarte mare ajutor în compilarea și rularea aplicațiilor de dimensiuni mai mari.

Anexa B este dedicată convențiilor de scriere a programelor. Sunt prezen-

tate principalele reguli de scriere a unor programe lizibile, conforme cu standardul stabilit de Sun Microsystems. Ultima parte a anexei este dedicată unei unelte foarte utile în documentarea aplicațiilor Java: *javadoc*.

Anexa C detaliază ideea de pachete, oferind informații despre pachete Java predefinite și despre posibilitatea programatorului de a defini propriile sale pachete. Un accent deosebit se pune și pe prezentarea arhivelor *jar*.

Anexa D prezintă modul în care se pot realiza aplicații internaționalizate, prin care textele care apar într-o aplicație sunt traduse dintr-o limbă în alta, cu un efort minim de implementare. De asemenea, este prezentat și rolul colecțiilor de resurse (engl. resource bundles) în internaționalizarea aplicațiilor.

Anexa E reprezintă o listă de resurse disponibile programatorului Java, pornind de la *site*-ul Sun Microsystems și până la tutoriale, cărți, reviste online, liste de discuții disponibile pe Internet. Cu ajutorul acestora, un programator Java poate să își dezvolte aptitudinile de programare și să acumuleze mai multe cunoștințe despre limbajul Java.

Volumul al doilea este destinat prezentării algoritmilor. Independența algoritmilor relativ la un anumit limbaj de programare, face ca majoritatea programelor din această parte să fie realizate și în pseudocod, punctând totuși pentru fiecare în parte specificul implementării în Java.

Capitolul 9 constituie primul capitol al celui de-al doilea volum și prezintă modalitatea prin care se poate realiza analiza eficienței unui algoritm. Notăția asimptotică, tehnicile de analiză a algoritmilor, algoritmi recursivi constituie principala direcție pe care se axează acest capitol.

Capitolul 10 reprezintă o incursiune în cadrul structurilor de date utilizate cel mai frecvent în conceperea algoritmilor: stive, cozi, liste înlanțuite, arbori binari de căutare, tabele de repartizare și cozi de prioritate. Fiecare dintre aceste structuri beneficiază de o prezentare în detaliu, însoțită de o implementare Java în care se pune accent pe separarea interfeței structurii de date de implementarea acesteia.

Capitolul 11 constituie startul unei suite de capitole dedicate metodelor fundamentale de elaborare a algoritmilor. Primul capitol din această suită este rezervat celei mai elementare metode: *backtracking*. După o analiză amănunțită a caracteristicilor acestei metode (cum ar fi cei patru pași standard în implementarea metodei: *atribuie și avansează, încercare eșuată, revenire, revenire după construirea unei soluții*), sunt prezentate câteva exemple de probleme clasice care admit rezolvare prin metoda *backtracking*: generarea permutărilor, a aranjamentelor și a combinațiilor, problema damelor, problema colorării hărților.

Capitolul 12 prezintă o altă metodă de elaborare a algoritmilor: *divide et impera*. Prima parte a capitolului este rezervată prezentării unor noțiuni intro-

ductive despre recursivitate și recurență, absolut necesare înțelegerii modului în care funcționează această metodă. Analog capitolului 11, prezentarea propriu-zisă a metodei este însoțită de câteva exemple de probleme clasice rezolvabile prin această metodă: căutarea binară, sortarea prin interclasare (*mergesort*), sortarea rapidă (*quicksort*), trecerea expresiilor aritmetice în formă poloneză postfixată.

Capitolul 13 prezintă cea de-a treia metodă de elaborare a algoritmilor: *metoda Greedy*. Capitolul păstrează aceeași structură ca și cele precedente: sunt prezentate mai întâi elementele introductive ale metodei, urmate apoi de câteva exemple clasice de probleme rezolvabile prin această metodă: problema spectacolelor, minimizarea timpului de așteptare, interclasarea optimă a mai multor șiruri ordonate.

Capitolul 14 este rezervat unei metode speciale de elaborare a algoritmilor: *programarea dinamică*, ce reprezintă probabil cea mai complexă metodă de elaborare a algoritmilor, punând deseori în dificultate și programatorii experimentați. Totuși avem credința că modul simplu și clar în care sunt prezentate noțiunile să spulbere mitul care înconjoară această metodă. Capitolul debutează cu o fundamentare teoretică a principalelor concepte întâlnite în cadrul metodei. Apoi, se continuă cu rezolvarea unor probleme de programare dinamică: înmulțirea unui șir de matrice, subșirul comun de lungime maximă, distanța Levenshtein etc.

Capitolul 15 reprezintă ultimul capitol din seria celor dedicate metodelor de elaborare a algoritmilor. Metoda *branch and bound* este cea abordată în cadrul acestui capitol, prin intermediul unui exemplu clasic de problemă: jocul de *puzzle* cu 15 elemente.

Capitolul 16 reprezintă o sinteză a metodelor de elaborare a algoritmilor care au fost tratate de-a lungul volumului al doilea, prezentând aspecte comune și diferențe între metode, precum și aria de aplicabilitate a fiecărei metode în parte.

Cui se adresează această carte?

Lucrarea de față nu se adresează începătorilor, ci persoanelor care stăpânesc deja, chiar și parțial, un limbaj de programare. Cititorii care au cunoștințe de programare în C și o minimă experiență de programare orientată pe obiecte vor găsi lucrarea ca fiind foarte ușor de parcurs. Nu sunt prezentate noțiuni elementare specifice limbajelor de programare cum ar fi funcții, parametri, instrucțiuni etc. Nu se presupune cunoașterea unor elemente legate de programarea orientată pe obiecte, deși existența lor poate facilita înțelegerea noțiu-

nilor prezentate. De asemenea, cartea este foarte utilă și celor care doresc să aprofundeze studiul algoritmilor și modul în care anumite probleme clasice de programare pot fi implementate în Java.

Pe Internet

Pentru comoditatea cititorilor, am decis să punem la dispoziția lor codul sursă complet al tuturor programelor prezentate pe parcursul celor două volume ale lucrării în cadrul unei pagini web concepută special pentru interacțiunea cu cititorii. De asemenea, pagina web a cărții va găzdui un forum unde cititorii vor putea oferi sugestii în vederea îmbunătățirii conținutului lucrării, vor putea schimba opinii în legătură cu diversele aspecte prezentate, adresa întrebări autorilor etc. Adresele la care veți găsi aceste informații sunt:

- <http://www.albastra.ro/carti/v178/>
- <http://www.danciu.ro/apj/>

Mulțumiri

În încheiere, dorim să adresăm mulțumiri colegilor și prietenilor noștri care ne-au acordat ajutorul în realizarea acestei lucrări: Vlad Petric (care a avut o contribuție esențială la structurarea capitolului 14), Alexandru Băluț (autor al anexei A), Iulia Tatomir (a parcurs și comentat cu multă răbdare de mai multe ori întreaga carte) și Raul Furnică (a parcurs și comentat capitolele mai delicate ale lucrării).

1. Prezentarea limbajului Java

Descoperirea constă în a vedea
ceea ce toată lumea a văzut și în a
gândi ceea ce nimeni nu a gândit.

Albert Szent-Gyorgi

În cadrul acestui capitol vom prezenta informații cu caracter general care să vă permită realizarea unei idei de ansamblu asupra limbajului Java. Sunt prezentate informații despre următoarele subiecte:

- De ce este important să învățați Java și de ce este Java un concurent serios pentru celelalte limbaje de programare;
- Cum a fost creat limbajul Java și care a fost evoluția sa în anii care au urmat de la apariția sa în 1995;
- Care sunt companiile care implementează limbajul Java;
- Prezentare detaliată a celei mai importante implementări a limbajului Java, cea oferită de Sun Microsystems, incluzând informații despre terminologia de bază utilizată;
- Ce cuprinde platforma Java oferită de Sun Microsystems.

1.1 Limbaje de programare

Probabil că primul lucru la care se gândesc cei care cunosc și alte limbaje de programare este la ce le folosește asimilarea unui nou limbaj? Nu sunt de ajuns cele pe care le cunosc deja? Oricât de simplu ar părea, un răspuns ferm la această întrebare nu este chiar atât de ușor de formulat. Deși sunt numeroase

limbaje de programare în lumea calculatoarelor, între ele există multe diferențe. Fiecare are avantaje și dezavantaje în raport cu celelalte. Nu există un limbaj de programare perfect care să înglobeze toate avantajele oferite de limbajele existente.

Deși numărul de limbaje de programare este relativ mare, limbajele folosite în dezvoltarea de aplicații software sunt în general aceleași. Java face parte din această categorie, pentru că este un limbaj de programare remarcabil, care ușurează sarcina programatorilor de a realiza aplicații de o complexitate incredibilă. Experiența în domeniul software a arătat că învățarea limbajului Java poate să îmbunătățească nivelul calitativ al unui programator. Pe lângă avantajele practice de a avea mai multe posibilități de a rezolva o problemă, învățarea acestui limbaj poate să lărgască perspectivele și modalitățile de abordare a problemelor care sunt dificil de implementat în celelalte limbaje. Să considerăm de exemplu cazul în care aveți deja experiență de programare în C sau C++. Este foarte dificil să creați un program Java fără să înțelegeți noțiunile de clase și obiecte, spre deosebire de C++, unde se poate folosi vechiul stil de programare procedural, specific limbajului C. A învăța Java presupune a învăța programare orientată pe obiecte (engl. *object oriented programming*, pe scurt OOP¹), ceea ce vă poate ajuta cu siguranță să deveniți un programator mai bun. Iar Java oferă un înțeles mult mai clar unor concepte mai complexe, cum este OOP, decât reușesc alte limbaje. De asemenea, în cazul multor probleme, veți putea scrie mai rapid o soluție în Java decât în C sau C++. Mai mult, datorită structurii sale, soluția Java va fi mai puțin predispusă erorilor (vezi problemele spinoase cauzate de operațiile cu *pointeri*).

Lucrarea de față urmărește să vă prezinte de-a lungul capitolelor sale într-un mod cât mai atractiv tehnologia Java și, totodată, să vă adâncească și mai mult convingerea că acest limbaj se află printre cele care oferă cele mai puternice facilități în domeniul software. Iată numai câteva dintre caracteristicile care vor fi studiate în amănunțime: tipuri de date primitive, operatori, metode, referințe, obiecte, clase, OOP, pachete, moștenire, interfețe, clase interioare, RTTI (RunTime Type Identification), mecanismul *reflection*, excepții, operații de intrare-ieșire (I/O), fire de execuție, convenții de scriere a programelor Java, *javadoc*, colecții de resurse, aplicații internaționalizate etc.

¹OOP = modalitate de a conceptualiza un program ca un set de obiecte aflate în interacțiune, având scopul de a îndeplini o sarcină.

1.2 Începuturile limbajului Java

Spre deosebire de alte limbaje de programare, este foarte probabil ca informațiile pe care le aveți despre tehnologia Java să nu fie mai vechi de câțiva ani. Motivul este simplu: Java a apărut doar cu câțiva ani în urmă, în 1995. 23 mai 1995 a fost ziua în care a fost lansată oficial tehnologia care a schimbat radical lumea calculatoarelor în cei aproape opt ani de la apariție. În acea zi, John Gage, directorul departamentului "Science Office" din cadrul Sun Microsystems, și Marc Andreessen, co-fondator și vicepreședinte al firmei Netscape, au anunțat în cadrul conferinței SunWorld că tehnologia Java a devenit realitate și că urmează să fie încorporată în Netscape Navigator, browser-ul pentru Internet al firmei Netscape. La acea dată, întreaga echipă de la Sun care concepușe Java avea mai puțin de 30 de membri.

Începuturile limbajului Java datează însă dinainte de 1995. Tehnologia Java a fost creată ca o unealtă de programare în cadrul unui proiect privat de dimensiuni reduse, inițiat în cadrul companiei Sun, în anul 1991, de către un mic grup care îi cuprindea printre alții pe Patrick Naughton, Mike Sheridan și James Gosling. Proiectul purta numele "The Green Project" și nu avea ca scop crearea unui nou limbaj de programare. Grupul, format din 13 membri, primise de la Sun Microsystems sarcina de a anticipa și a plănuși "noul val" în lumea tehnologiei informației (engl. IT). După o perioadă de timp în care au analizat factorii implicați, concluzia membrilor grupului a fost că o tendință importantă în următorii ani urma să fie dezvoltarea dispozitivelor electrocasnice "inteligente" - televizoare interactive, iluminat interactiv, etc. Cercetătorii Sun considerau că viitorul era rezervat dispozitivelor care puteau comunica între ele, în așa fel încât, de exemplu, televizorul să-i comunice telefonului mobil că a început emisiunea dumneavoastră preferată, iar acesta din urmă să vă alerteze printr-un semnal acustic. Pentru a demonstra ce văd ei ca fiind viitorul dispozitivelor digitale, cei 13 membri au prezentat în vara lui 1992, după 18 luni de muncă neîntreruptă, un dispozitiv interactiv cu interfață grafică, asemănător unui mic televizor portabil. În acea demonstrație, de acum cunoscută mascotă a tehnologiei Java, ducele ("The Duke"), executa câteva animații pe ecran. Prototipul era denumit *7 ("StarSeven") și se remarcă prin faptul că era capabil să comunice și cu alte dispozitive. *7 funcționa cu ajutorul unui limbaj cu totul nou. Inițial, ideea a fost de a realiza sistemul de operare al *7 în C++, ultrapopularul limbaj de programare orientat pe obiecte, dar până la urmă s-a decis crearea unui nou limbaj de programare, special pentru *7, care să comunice mai bine cu el. Limbajul a fost creat de James Gosling, membru al grupului, care l-a denumit "Oak" ("stejar"), după copacul din fața ferestrei sale. Mai târziu, Sun a descoperit că acest nume era deja folosit pentru un alt produs și l-a redenu-

mit în *Java*. Numele a fost ales după multe ore de gândire, dintr-o listă cu numeroase alte alternative. Ideea membrilor grupului era să găsească un nume care să cuprindă esența tehnologiei: vioiciune, animație, viteză, interactivitate etc. *Java* nu reprezenta un acronim, ci mai degrabă o amintire a acelui lucru fierbinte și aromat pe care mulți programatori îl beau în mari cantități (pentru cei care nu s-au prins, este vorba de cafea).

Fiind proiectat pentru a fi folosit de dispozitive electrocasnice și nu de calculatoare super-performante, *Java* trebuia să fie mic, eficient și să funcționeze pe o gamă largă de dispozitive hardware. De asemenea, trebuia să fie fiabil. Utilizatorii de calculatoare s-au mai obișnuit cu clasicele "căderi" ale sistemului, însă această situație nu putea fi acceptabilă pentru utilizatorii de aparatură electrocasnică (majoritatea utilizatorilor de aparatură electrocasnică au dificultăți în a învăța cum să manipuleze corect noul aparat - cum ar fi dacă acesta ar mai și reacționa uneori complet neprevăzut).

Pe măsură ce în cadrul proiectului s-au implicat potențiali clienți, acesta a devenit public, iar grupul și-a schimbat numele din "The Green Team" în "FirstPerson". Cu timpul, echipa "FirstPerson" a început să caute o piață de desfacere pentru dispozitivul *7, iar industria TV părea să fie cea mai potrivită pentru acest lucru. Echipa a creat un nou demo, denumit "MovieWood", pentru a demonstra potențialul tehnologiei pe piața dispozitivelor TV. Din păcate, industria televizoarelor digitale se afla în perioada de început și proiectul nu a mai înaintat în această direcție. Următorul pas a fost încercarea de a convinge companiile de televiziune prin cablu să utilizeze noua tehnologie. Utilizarea ei ar fi transformat aceste companii în adevărate rețele interactive, asemănătoare Internet-ului din zilele noastre, rețele în care utilizatorii ar fi putut citi sau scrie informații, dar companiile de cablu au considerat că ar putea pierde controlul asupra rețelelor, tentativa de promovare fiind și de această dată sortită eșecului.

Curând ei au realizat că noua tehnologie nu poate pătrunde în industria TV și s-au decis să încerce Internetul, care la acea vreme de abia începuse să câștige în popularitate. Internetul reprezenta exact tipul de rețea pe care echipa îl imaginase pentru industria TV și a televiziunilor prin cablu. Internetul devenise un mijloc popular de a transporta informații multimedia (text, imagini, filme) în cadrul unei rețele și se asemana din acest punct de vedere cu tehnologia *Java*. Deși Internetul avea deja 20 de ani de când apăruse, fusese dificil de folosit până la apariția programului *Mosaic* în 1993, pentru că permitea doar folosirea tehnologiilor *ftp* (FileTransferProtocol) sau *telnet*. Apariția *Mosaic* a revoluționat modul în care oamenii percepeau Internetul, deoarece avea o interfață grafică foarte ușor de utilizat. A fost momentul în care membrii grupului au ajuns la concluzia că *Java* trebuie să devină o tehnologie destinată Internetului.

Primul pas făcut de grup în această direcție a fost crearea unei clone a *Mo-*

saic, bazată pe tehnologie Java. Aplicația a fost denumită *WebRunner* (după filmul "Blade Runner"), dar a devenit oficial cunoscută sub numele de *HotJava*. Era 1994, anul apariției primului *browser* bazat pe tehnologie Java. Deși era la stadiul de aplicație demo, *HotJava* era impresionant: pentru prima dată un browser de Internet oferea conținut animat. La începutul anului 1995, Gosling și Gage au făcut o primă prezentare publică a noului browser *HotJava*, care a avut un succes răsunător. Astfel că la începutul lunii martie 1995, versiunea alfa (denumită "1.0a2") a produsului *HotJava* a devenit publică pe Internet. Codul sursă al aplicației a fost făcut și el public. În săptămânile care au urmat, numărul de download-uri al aplicației *HotJava* a crescut impresionant, atingând 10.000 de copii. A fost un număr care a depășit cu mult până și așteptările grupului. Trăsăturile care au făcut Java să fie bun pentru *7 s-au dovedit a fi bune și pentru Internet. Java era:

- mic - programe de dimensiuni reduse ofereau multe facilități;
- portabil - putea fi rulat pe diferite platforme (Windows, Solaris, Linux, etc.) fără modificări;
- sigur - împiedica scrierea de programe care produceau pagube calculatoarelor.

Curând, Sun a realizat că popularitatea tehnologiei Java și a grupului care o crease, a atins un nivel foarte ridicat și a trecut la fapte.

Pe 23 mai 1995, tehnologia Java a fost lansată oficial de Sun Microsystems. Evenimentele nu s-au oprit însă aici. În aceeași zi, Eric Schmidt și George Paolini din cadrul Sun Microsystems împreună cu Marc Andreessen din cadrul Netscape au semnat înțelegerea conform căreia tehnologia Java urma să fie integrată în omniprezentul și omnipotentul (la acea vreme...) browser Netscape Navigator. Surpriza era cu atât mai mare cu cât nici membrii echipei "FirstPerson" nu știau nimic despre tratativele dintre cele două părți.

Cu toate că Java și *browser*-ul HotJava s-au bucurat de o mare atenție din partea comunității Web, limbajul s-a lansat cu adevărat abia după ce Netscape a fost prima companie care l-a licențiat. Imediat după prima lansare, Sun și-a îndreptat eforturile de dezvoltare Java printr-o nouă sucursală: JavaSoft. De-a lungul anilor care au urmat, tehnologia Java a câștigat foarte mult în maturitate, iar potențialul ei este încă enorm. Au apărut JDK-ul, applet-urile, mii de cărți despre tehnologia Java, arhitectura JavaBeans, Netscape Communicator, servlet-urile, Java Server Pages (JSP), Java Foundation Classes (JFC), componentele Enterprise JavaBeans (EJB), comerțul electronic, precum și implicarea în dezvoltarea tehnologiei Java a unor firme deosebit de importante, cum ar fi

IBM. Toate aceste lucruri arată cât de rapid s-a dezvoltat tehnologia Java în anii care au trecut de la lansarea ei oficială. Java este acum un mediu important de dezvoltare a aplicațiilor. Foarte mulți utilizatori au fost atrași de obiectivul pe care această tehnologie îl are: *"write once, run anywhere"* ("scris o dată, rulat oriunde"), ceea ce tradus înseamnă că folosind tehnologia Java, același program devine portabil și poate fi executat pe mai multe sisteme de operare/platforme (de exemplu: Windows, Linux, Mac, Solaris etc.). De la apariția ei în mai 1995, platforma Java a fost adoptată în industria IT mai rapid decât orice altă nouă tehnologie din istoria calculatoarelor. Se poate spune că Java a devenit limbajul, platforma și arhitectura calculului în rețea.

1.3 Caracteristici ale limbajului Java

Limbajul Java se bucură de o imensă popularitate în rândul programatorilor. Puterea sa constă în faptul că este:

- independent de platformă;
- orientat pe obiecte;
- ușor de învățat și de folosit în dezvoltarea de aplicații software;
- disponibil gratuit, factor care a dus la rapida sa răspândire.

Posibilitatea ca aceeași aplicație să ruleze nemodificată pe diferite platforme (sisteme de operare) este unul dintre cele mai semnificative avantaje pe care limbajul Java le are asupra altor limbaje de programare. După cum se știe, atunci când se compilează un program C (sau în alt limbaj de programare), compilatorul transformă fișierul sursă² în *cod-mașină* - instrucțiuni specifice sistemului de operare și procesorului pe care îl folosește calculatorul respectiv. Dacă se compilează codul sursă pe un sistem Windows, programul va rula pe un sistem Windows, dar nu va putea rula pe un sistem Linux. Pentru a-l rula pe Linux, codul sursă trebuie transferat pe Linux și recompilat, pentru a produce cod executabil pe noul sistem. În concluzie, pentru fiecare sistem trebuie produs câte un program executabil separat. Programele Java dobândesc independență din punct de vedere al sistemului de operare prin folosirea unei *mașini virtuale*³ - un fel de calculator în alt calculator. Mașina virtuală preia programul

²denumit și *cod sursă*, reprezintă un set de instrucțiuni pe care un programator le introduce cu ajutorul unui editor de texte, atunci când crează un program

³denumită JVM, sau interpretor Java

Java compilat și transformă instrucțiunile în comenzi inteligibile sistemului respectiv. Acest program compilat (într-un format denumit *bytecode*⁴) poate rula pe orice platformă (sistem de operare) care posedă o mașină virtuală Java. Și nu este vorba numai de PC-uri, ci și de alte dispozitive, cum ar fi: telefoane mobile, PDA-uri etc. Astfel, programatorii Java nu sunt nevoiți să creeze versiuni diferite ale programelor pentru fiecare platformă pe care folosesc aplicația, deoarece mașina virtuală este cea care realizează “traducerea” necesară. Mașina virtuală nu adaugă un nivel nedorit între sursă și codul-mașină, fiind indispensabilă pentru a asigura independența de platformă.

Orientarea pe obiecte a limbajului Java este o caracteristică logică, ținând cont de faptul că Java își are rădăcinile în limbajul C++, moștenind de la acesta majoritatea conceptelor OOP.

Java a fost modelat foarte asemănător cu C++, majoritatea sintaxei și a structurii obiect orientate provenind direct din acest limbaj. Pentru un programator C++ este foarte ușor să învețe Java. În ciuda asemănării cu C++, noțiunile cele mai complexe (și totodată cele mai mari generatoare de erori) au fost eliminate în Java: pointeri, aritmetică cu pointeri, gestionarea memoriei se face automat (și nu de către programator). Un program Java este ușor de scris, compilat și depanat.

1.4 Implementări ale limbajului Java

Tehnologia Java poate fi împărțită în patru componente:

- Mașina virtuală, denumită *JVM* (Java Virtual Machine), interpretează fișierele cu extensia *.class* care conțin bytecode. Bytecode-ul reprezintă cod compilat care este procesat de un program, denumit *mașină virtuală*, spre deosebire de codul executabil normal care rulează pe mașina *reală* care dispune de un procesor hardware. Bytecode-ul este format din instrucțiuni generice care vor fi apoi convertite de către mașina virtuală în instrucțiuni specifice procesorului. Acesta este motivul pentru care aplicațiile Java sunt independente de platformă. Aplicațiile Java nu trebuie modificate pentru a le putea rula pe platforme diferite de cea pe care au fost create. Este sarcina mașinii virtuale de pe respectiva platformă să creeze cod executabil specific platformei. Cu alte cuvinte, o aplicație

⁴codul mașină pentru mașina virtuală, adică instrucțiuni pe care mașina virtuală le înțelege direct. Codul sursă este compilat în *bytecode*, astfel încât poate fi rulat pe o mașină virtuală Java.

Java scrisă sub Windows poate fi rulată și pe Linux, fără a fi necesară nici măcar o recompilare. Interpretarea byte-code-ului rămâne în sarcina mașinilor virtuale de pe cele două platforme, în cazul nostru Windows și Linux. Pentru a putea rula aplicații Java, transformate prin compilare în fișiere *.class*, este absolut necesară o mașină virtuală. Lipsa acesteia duce la imposibilitatea execuției aplicațiilor Java;

- Limbajul Java propriu-zis. Limbajul Java este orientat pe obiecte și se aseamănă din punct de vedere al sintaxei cu C++, simplificat însă pentru a elimina elementele de limbaj cauzatoare de erori de programare;
- Un compilator care generează fișierele cu extensia *.class*. Sarcina compilatorului este de a crea, pornind de la programul scris în limbajul Java (adică, de la fișiere cu extensia *.java*), fișierele *.class*, care vor fi interpretate de mașina virtuală;
- Biblioteca de clase Java, denumită și Java *API* (*engl.* Application Programming Interface). Tehnologia Java include un set de componente puternice și utile, care pot fi reutilizate de programatori în aplicații de natură foarte diversă.

Terminologia Java este uneori destul de confuză. Multe persoane, inclusiv cei de la Sun, folosesc termenul “*Java*” pentru fiecare dintre aceste componente, dar și pentru tehnologia în ansamblu. Din acest motiv, înțelesul termenilor trebuie adeseori desprins din context. De exemplu, atunci când cineva spune că rulează un program Java, prin aceasta trebuie să înțelegem că de fapt rulează un set de fișiere *.class* pe mașina virtuală *JVM*.

Orice implementare a limbajului Java presupune implicit implementarea tuturor celor patru componente prezentate anterior. Este și cazul celor mai importante implementări ale limbajului Java, realizate de două nume mari în lumea calculatoarelor: Sun și IBM. Implementările oferite de cele două mari firme nu sunt singurele. De o importanță mai redusă sunt: Blackdown JDK (disponibil pe Linux, detalii la adresa <http://www.blackdown.org>), Kaffe (disponibil pe Linux, distribuit cu Linux-ul în sine, detalii la adresa: <http://www.kaffe.org>).

Nici Microsoft nu a rămas datoare la acest capitol și a creat propria sa implementare Java, disponibilă (evident) numai pe platformele Windows. Implementarea poartă numele de *Microsoft SDK for Java* și cuprinde un compilator Java și biblioteca de clase, mașina virtuală fiind download-abilă separat. Merită semnalat aici, ca o paranteză, faptul ca această implementare a limbajului Java a fost obiectul unui proces între Microsoft și Sun. După trei ani procesul

a luat sfârșit, în ianuarie 2001. Sentința nu a fost o surpriză pentru lumea IT, pentru că șansele ca Microsoft să câștige erau reduse. Motivul procesului a fost faptul că Microsoft a încălcat în mod intenționat contractul cu Sun, plasând în mod repetat logo-ul Java deținut de Sun pe aplicații software care nu satisfăceau standardele de compatibilitate impuse de Sun prin contractul dintre cele două părți. Există păreri care susțin că Microsoft nici nu a dorit să satisfacă standardul de compatibilitate, pentru că astfel produsele sale ar fi devenit disponibile și pe platformele non-Microsoft. Ca rezultat al sentinței judecătorești, Microsoft a plătit 20 de milioane de dolari companiei Sun, nu va mai primi licență Java, nu va mai putea folosi logo-ul Java și va putea să distribuie doar versiunea 1.1.14 a implementării Java, dar și aceasta sub o licență limitată, care va expira peste șapte ani. Ca răspuns, Microsoft încearcă să convingă dezvoltatorii de aplicații software să folosească platforma proprietară “.NET”, împreună cu propriul limbaj de programare C#. Rămâne de văzut însă ce succes va avea această platformă.

Dintre implementările Java, cel mai frecvent utilizată este cea realizată de Sun, acesta fiind și motivul pentru care o vom descrie mai detaliat în paragraful 1.5. Există situații în care programatorii optează pentru implementarea IBM, în special în cazul în care Sun nu oferă o implementare Java pentru platforma dorită. Din acest motiv, se cuvine să prezentăm pe scurt și implementarea celor de la IBM.

IBM oferă implementări ale limbajului Java pentru următoarele platforme (în paranteze este trecută versiunea la care a ajuns implementarea):

- AIX (versiunea 1.3) ;
- Linux (versiunea 1.3);
- AS/400;
- OS/2 (versiunea 1.3);
- OS/390 (versiunea 1.1.8);
- VM/ESA (versiunea 1.1.6);
- Windows (versiunea 1.1.8).

Cu excepția implementărilor pentru Windows și Linux, celelalte implementări ale IBM nu sunt disponibile la Sun (în plus față de IBM, Sun realizează și o implementare Java pentru Solaris), așa că dacă platforma pe care o folosiți este una dintre acestea, va trebui să apelați la implementarea IBM corespunzătoare.

Un alt avantaj al implementărilor IBM este compilatorul *jikes*, compilatorul Java realizat de IBM. Acesta este mai rapid și mai "prietenos" cu utilizatorul, decât compilatorul oferit de Sun.

Implementările IBM trec printr-un proces complex de validare de la realizare și până la lansarea către publicul larg. Pentru a fi făcute publice, ele trebuie să fie stabile și să treacă testul de compatibilitate al firmei Sun.

1.5 Implementarea Sun a limbajului Java

Platforma Java se bazează pe arhitectura de rețea și pe ideea că aceeași aplicație software trebuie să ruleze pe diferite tipuri de calculatoare, dar și pe alte dispozitive (de exemplu, sistemul de navigare al unui automobil). De la lansarea ei comercială, în mai 1995, tehnologia Java a crescut în popularitate tocmai datorită portabilității ei. Platforma Java permite executarea aceleiași aplicații Java pe diferite tipuri de calculatoare. Orice aplicație Java poate fi ușor transmisă prin Internet sau prin orice tip de rețea, fără probleme de compatibilitate între sisteme de operare sau hardware. De exemplu, se poate rula o aplicație bazată pe tehnologie Java pe un PC, pe un calculator Macintosh, sau chiar pe anumite tipuri de telefoane mobile.

Tehnologia Java permite programatorilor și utilizatorilor să realizeze lucruri care înainte nu erau posibile. Prin intermediul Java, Internetul și rețelele de calculatoare devin mediul natural de dezvoltare al aplicațiilor. De exemplu, utilizatorii pot accesa în deplină securitate informații personale sau aplicații, utilizând un calculator conectat la Internet. De asemenea, ei pot accesa aplicații și de pe un telefon mobil bazat pe tehnologie Java sau pot folosi carduri *Smart*, de dimensiunea unei cărți de credit, ca modalitate de identificare.

1.5.1 Platformele Java

Datorită faptului că tehnologia Java a evoluat în mai multe direcții simultan, Sun a grupat tehnologia Java în trei ediții:

- Java 2 Platform, Micro Edition (platforma J2ME);
- Java 2 Platform, Standard Edition (platforma J2SE);
- Java 2 Platform, Enterprise Edition (platforma J2EE).

Fiecare dintre cele trei ediții oferă unelte necesare creării de aplicații software: o mașină virtuală Java adaptată resurselor fizice (memorie, procesor, etc.) ale

dispozitivului pe care rulează, aplicații utilitare pentru compilare și executare, și o bibliotecă de clase (API-ul) specializată pentru fiecare tip de dispozitiv.

Edițiile se adresează programatorilor care doresc să realizeze un anumit tip de aplicație software:

- *J2SE* reprezintă mediul de dezvoltare al aplicațiilor de tip *client-side* (care rulează pe mașina client, cum ar fi appleturile). *J2SE* este o unealtă de bază pentru crearea de aplicații sofisticate și puternice, datorită uneltelor de dezvoltare și ierarhiei de clase (*API*) pe care o oferă;
- *J2EE* reprezintă mediul de dezvoltare al aplicațiilor de tipul *server-side* (care rulează pe un server și pot fi accesate de mii de clienți simultan, cum ar fi servlet-urile sau Enterprise Java Beans). Tehnologia *J2EE* ușurează enorm crearea de aplicații internet industriale scalabile prin utilizarea de componente *Enterprise JavaBeans* standardizate, modulare și reutilizabile, și automatizând multe detalii de comportament ale aplicației, oferind astfel programatorilor mai mult timp pentru a dezvolta partea logică a aplicației fără să piardă timp cu partea de infrastructură;
- *J2ME* se adresează programatorilor care doresc să dezvolte aplicații pentru dispozitive cu resurse mai reduse decât un calculator PC obișnuit. Printre aceste dispozitive putem enumera: carduri *Smart*, *pager*-e, telefoane mobile, *PDA*-uri (*engl.* Personal Digital Assistant), anumite tipuri de televizoare.

Având în vedere faptul că fiecare dintre aceste trei ediții presupune o imensă cantitate de informații de acumulat, lucrarea de față va prezenta doar tehnologia *J2SE*.

1.5.2 Platforma J2SE (Java 2 Platform, Standard Edition)

În 1995, când tehnologia Java a fost oficial lansată, produsul Sun care permitea crearea și executarea programelor Java se numea *JDK* (Java Development Kit). În decembrie 1998, Sun a decis să împartă tehnologia Java în trei direcții și a fost necesară introducerea unor nume noi pentru tehnologiile nou create. Așa s-a ajuns la denumirea *Java 2*. Prin urmare, platforma *JDK* a fost redenumită *J2SE* (Java 2 Platform, Standard Edition). Celelalte două platforme, *J2EE* și *J2ME*, fiind nou apărute, nu a fost necesară redenumirea lor. La momentul apariției noii denumiri, produsul *JDK* era la versiunea 1.2, astfel că el a fost redenumit la rândul său *Java 2 SDK, Standard Edition, v 1.2* (*SDK* = Software Development Kit).

Probabil că aceste denumiri au creat deja o anumită stare de confuzie. În primul rând trebuie făcută diferența dintre *platformă* și *produs*. Prin *platformă* se înțelege o funcționalitate abstractă, iar prin *produs* se înțelege aplicația software care implementează platforma respectivă. Până în decembrie 1998, platforma Java se numea *JDK*, iar produsele se numeau *JDK 1.0*, *JDK 1.1*, în funcție de versiunea la care ajunsese produsul. În decembrie 1998, sau altfel spus, începând cu versiunea 1.2 a produsului *JDK*, platforma a fost denumită *Java 2 Platform, Standard Edition, v 1.2 (J2SE)*, iar produsul a fost denumit *Java 2 SDK, Standard Edition, v 1.2 (J2SDK)*. Noua denumire se aplică doar produselor *JDK* de la versiunea 1.2. Versiunile anterioare (*JDK 1.0*, *JDK 1.1.x*) își păstrează denumirea inițială. Un alt lucru care trebuie remarcat este că denumirea *Java 2* nu afectează denumirea mașinii virtuale, care este tot *JVM*, sau a limbajului în sine, care este tot *Java*.

Așadar, *J2SDK* este produsul Sun care implementează platforma abstractă *J2SE* și care este folosit pentru dezvoltarea de aplicații specifice platformei *J2SE*. Produsul include uneltele de creare a programelor Java (compilator etc.) și uneltele de execuție ale acestora (interpretor, mașina virtuală etc.). Componentele necesare pentru a executa programele Java sunt grupate sub forma unui subprodus, denumit *Java 2 Runtime Environment, Standard Edition (J2RE)*. *J2RE* reprezintă mediul în care se execută aplicațiile scrise pentru platforma *J2SE*. El este un produs Sun de sine stătător și poate fi folosit și fără *J2SDK*, dar în această situație nu se vor putea crea aplicații Java, ci doar rula unele deja create. Deci, dacă doriți doar să executați aplicații Java, nu este obligatoriu să instalați pe calculatorul dumneavoastră produsul *J2SDK*, fiind de ajuns să aveți *J2RE*. În schimb, dacă doriți să creați propriile aplicații Java, este absolut necesar să aveți *J2SDK*, pentru că *J2SDK* este produsul care cuprinde uneltele de dezvoltare a aplicațiilor Java.

Pentru a vă crea o imagine și mai bună asupra noțiunii de *J2SE*, vom prezenta în continuare conținutul acestei platforme, folosind ca exemplu versiunea curentă a platformei, la momentul redactării acestui paragraf (aprilie 2002). Pentru a fi la curent cu ultimele versiuni ale tuturor platformelor Java oferite de Sun, puteți afla detalii la adresa de Internet <http://java.sun.com>.

Versiunea curentă a platformei *J2SE* este 1.4 și a fost lansată în primăvara anului 2002. *Java 2 Platform, Standard Edition v 1.4*, denumirea completă a ultimei versiuni a platformei *J2SE*, este formată din:

- *Java 2 SDK, Standard Edition, v 1.4 (SDK)*;
- *Java 2 Runtime Environment, Standard Edition, v 1.4 (JRE)*;
- *Java 2 Platform, Standard Edition, v 1.4 Documentation (DOCS)*.

În continuare, vom prezenta în detaliu fiecare dintre cele trei componente ale J2SE.

1.5.3 J2SDK (Java 2 SDK, Standard Edition)

J2SDK, versiunea 1.4, este disponibilă pentru următoarele platforme (sisteme de operare): Windows, Linux și Solaris. Sun nu oferă implementări ale limbajului Java pentru alte sisteme de operare. Prin urmare, dacă sistemul de operare pe care îl folosiți diferă de cele trei enumerate mai sus, va trebui să găsiți o altă companie care să ofere o implementare a limbajului Java pentru platforma dumneavoastră.

Java 2 SDK reprezintă mediul de dezvoltare pentru aplicații, applet-uri și componente la standarde profesionale, folosind limbajul de programare Java. Java 2 SDK include unelte utile pentru dezvoltarea și testarea programelor scrise în limbajul de programare Java și executate pe platforma Java. Aceste unelte sunt proiectate pentru a fi folosite în linie de comandă (de exemplu, promptul MS-DOS pe Windows), cu alte cuvinte, nu au interfață grafică. Deși acest mod de operare pare într-un fel de modă veche, el reprezintă totuși o metodă foarte eficientă pentru utilizatorii obișnuiți cu aceste aplicații utilitare. Cei mai mulți utilizatori de PC-uri asociază utilizarea liniei de comandă cu sistemul de operare MS-DOS. Totuși, uneltele oferite de J2SDK nu sunt aplicații create pentru sistemul de operare MS-DOS. De altfel, nici nu există vreo versiune de J2SDK/JDK destinată sistemului MS-DOS. Decizia de utilizare a aplicațiilor utilitare în linie de comandă a aparținut dezvoltatorilor de programe Java. Pe de altă parte, faptul că sunt folosite aplicații în linie de comandă pentru a compila programele Java, nu înseamnă că nu se pot crea aplicații cu interfață grafică. Din contră, mediul de dezvoltare Java oferă facilități sofisticate pentru interfața grafică, de exemplu applet-urile, Swing, AWT (Abstract Window Toolkit) sau JFC. Aceste componente nu vor face însă subiectul lucrării de față. Mai mult, există nenumărate editoare și medii de programare pentru Java, care automatizează procesul compilării. O prezentare a celor mai des utilizate o veți găsi în cadrul anexei A.

După cum spuneam, produsul J2SDK oferă o multitudine de aplicații utilitare pentru compilarea, executarea, verificarea și managementul programelor Java. Dintre acestea, trei sunt cele mai importante:

- `javac`, compilatorul Java. Este folosit pentru a transforma programele Java într-o formă care poate fi executată;
- `java`, interpretorul Java. Este folosit pentru a rula programele Java compilate cu `javac`;

- `appletviewer`, folosit la testarea applet-urilor. Este folosit pentru a executa applet-urile fără a folosi un browser de Internet. Această aplicație este singura dintre aplicațiile utilitare care oferă interfață grafică, deși se lansează în linie de comandă ca și celelalte unelte.

Pe lângă aceste aplicații utilitare, J2SDK mai cuprinde:

- mașina virtuală JVM;
- biblioteca de clase (API-ul);
- documentația J2SDK (disponibilă separat).

Documentația J2SDK este de o importanță covârșitoare pentru orice programator, de aceea trebuie să insistăm mai mult asupra acestui subiect. Documentația poate fi accesată *online* pe site-ul firmei Sun, dar poate fi și descărcată pentru a fi utilizată în cazul în care nu există o conexiune Internet disponibilă permanent (cu alte cuvinte, pentru utilizare *offline*). Sun oferă o gamă largă de documentații: specificațiile API, ghidul programatorului, tutoriale, cărți, etc. Cele mai multe dintre aceste documente sunt în format HTML (HyperTextMarkupLanguage), ps (PostScript) sau pdf (PortableDataFormat).

Specificația API prezintă toate clasele din biblioteca de clase pe care J2SDK o oferă. Tot acolo este descrisă funcționalitatea fiecărei clase și sunt oferite exemple de utilizare în anumite contexte. Practic, această specificație este documentul cel mai utilizat de un programator Java.

Suplimentar, cei interesați mai pot găsi:

- aplicații demonstrative împreună cu codul sursă, disponibile în directorul `demo` al distribuției produsului J2SDK;
- tutorialul Java ("The Java Tutorial") realizat de Sun, care cuprinde sute de exemple complete și poate fi descărcat de la adresa <http://java.sun.com/docs/books/tutorial/information/download.html>;
- o listă de cărți utile, disponibilă la adresa <http://java.sun.com/docs/books>.

Informații complete despre documentația J2SDK, v 1.4 se pot afla la adresa de Internet: <http://java.sun.com/j2se/1.4/docs/index.html>.

Pentru a obține produsul J2SDK, trebuie să accesați pagina de Internet <http://java.sun.com/j2se>. De acolo, puteți descărca produsul compatibil cu sistemul de operare pe care îl utilizați. Tot de acolo veți putea descărca și documentația J2SDK.

Procedura de instalare a platformei J2SDK crează o ierarhie de directoare care va fi detaliată în cele ce urmează. Pentru a putea verifica autenticitatea informațiilor prezentate, este necesar să instalați J2SDK pe calculatorul dumneavoastră.

În urma instalării, distribuția Java 2 SDK conține:

- aplicații utilitare pentru dezvoltarea de programe Java (în subdirectorul `bin`). Aceste unelte ajută programatorul să creeze, să execute, să verifice și să documenteze programe scrise în limbajul Java;
- mediul de execuție (în subdirectorul `jre`), format din mașina virtuală Java, biblioteca de clase și alte fișiere necesare execuției de programe scrise în limbajul Java. Acest mediu reprezintă o implementare internă a mediului de execuție Java și nu trebuie confundat cu *Java 2 Runtime Environment (J2RE* - paragraful 1.5.4). Instalarea J2SDK are ca rezultat și instalarea J2RE (pentru a verifica acest lucru în cazul sistemului de operare Windows, accesați directorul `JavaSoft`, aflat în directorul `Program Files`, în cazul unei instalări standard);
- alte biblioteci (în subdirectorul `lib`), necesare uneltelor de dezvoltare pentru a funcționa corect;
- aplicații demonstrative (în subdirectorul `demo`), ca exemple de programare în limbajul Java;
- fișiere *header C* (în subdirectorul `include`), pentru uz intern;
- codul sursă (în fișierul `src.jar`), pentru o parte a claselor care formează API-ul Java. Acest cod sursă este oferit de Sun doar în scopuri informative, pentru a ajuta programatorii să învețe limbajul de programare Java. Codul dependent de platformă nu este introdus în această arhivă de fișiere sursă. Pentru a vizualiza fișierele, trebuie să dezarhivați fișierul, folosind programul `jar` aflat în distribuția J2SDK. Comanda trebuie executată în linie de comandă, în directorul în care se află fișierul `src.jar` și arată astfel:

```
jar xvf src.jar.
```

În urma executării acestei comenzi se va crea un director `src`, care va conține fișierele sursă.

J2SDK reprezintă primul instrument de dezvoltare de aplicații care suportă noile versiuni ale limbajului Java, adesea cu șase luni înaintea apariției altui software de dezvoltare Java (de exemplu mediile de programare Symantec Visual Cafe, Borland JBuilder, Metrowerks CodeWarrior, etc).

1.5.4 J2RE (Java 2 Runtime Environment, Standard Edition)

Java 2 Runtime Environment este produsul prin intermediul căruia se pot executa aplicații scrise în limbajul Java. El este destinat celor care doresc să poată rula aplicații Java, fără a avea nevoie de întreg mediul de dezvoltare. Ca și J2SDK, J2RE conține mașina virtuală Java (JVM) și biblioteca de clase care formează Java 2 API. Spre deosebire de J2SDK, J2RE nu conține unelte de dezvoltare, cum ar fi compilatorul.

J2RE este un produs care poate fi instalat separat, dar și ca o componentă a J2SDK. J2RE poate fi obținut la adresa:

<http://java.sun.com/j2se/1.4/jre/index.html>

Produsul poate fi redistribuit gratuit împreună cu o aplicație dezvoltată cu J2SDK, astfel încât utilizatorii respectivei aplicații să aibă o platformă Java pe care să o poată executa.

Sun oferă produsul J2RE pentru următoarele platforme: Windows, Linux și Solaris. Există o excepție în cazul Windows: J2RE nu funcționează pe platforma Windows NT 3.51.

1.5.5 Documentația J2SE (Java 2 Standard Edition Documentation)

Documentația J2SE este organizată pe mai multe categorii:

- tutorialul Java ("The Java Tutorial"), ghidul programatorilor Java, cuprinzând sute de exemple funcționale;
- documentația J2SDK;
- specificații ale limbajului Java și ale mașinii virtuale JVM.

Toate aceste documentații pot fi obținute de la adresa:

<http://java.sun.com/docs/index.html>

1.5.6 Licența de utilizare a platformei J2SE

Marea majoritate a produselor Sun pot fi obținute și folosite gratuit, chiar și pentru uz comercial. J2SDK și J2RE intră și ele în această categorie. Mai mult decât atât, ele pot fi redistribuite gratuit, cu condiția de a fi însoțite de o aplicație. Fiecare produs are propria lui licență de utilizare, deci, dacă doriți amănunte despre acest lucru, puteți verifica separat licența pentru fiecare produs.

Rezumat

Deși ați parcurs doar primul capitol al cărții, ați acumulat totuși o serie de informații despre istoria limbajului Java, principalele versiuni ale limbajului Java existente pe piață la ora actuală, și, nu în ultimul rând, v-ați familiarizat cu terminologia specifică. Din moment ce noțiunile de JDK, J2SE, `javac`, `java` vă sunt clare, putem trece la următoarea etapă a inițierii în tainele acestui limbaj. Pentru aceasta trebuie să vă instalați mediul de programare Java și pe calculatorul dumneavoastră. În momentul în care toate uneltele de dezvoltare a aplicațiilor Java (editoare, compilatoare, interpretoare) sunt la îndemâna dumneavoastră, vom putea realiza primele programe Java.

Noțiuni fundamentale

API: set de clase utile oferite împreună cu limbajul Java, în vederea reutilizării lor de către programatori;

J2RE: produs ce cuprinde toate aplicațiile ce permit rularea programelor Java. Reprezintă o componentă a J2SDK;

J2SDK: implementare a platformei abstracte J2SE, cu ajutorul căreia se pot crea aplicații Java;

J2SE: platformă abstractă pentru crearea de aplicații Java de tipul *client-side*;

`java`: interpretorul Java ce rulează aplicațiile Java compilate cu `javac` sau `jikes`;

`javac`: compilator Java realizat de Sun Microsystems;

JDK: vechea denumire a J2SDK, utilizată până în decembrie 1998;

`jikes`: compilator Java realizat de IBM;

JVM: mașina virtuală Java, ce interpretează fișierele de tip `.class`.

Erori frecvente

1. J2SDK este confundat deseori cu J2SE. Deși diferența dintre ele nu este foarte ușor de sesizat, totuși ea există. J2SE reprezintă o platformă abstractă (adică o specificație, un standard) în timp ce J2SDK este o implementare a platformei abstracte, este unealta cu ajutorul căreia se crează programele Java obișnuite. Până în decembrie 1998, J2SDK a fost cunoscut sub denumirea JDK.

2. Noțiuni fundamentale de programare în Java

Nu sunt nici deosebit de inteligent
și nici nu am vreun talent
remarcabil.

Sunt doar foarte, foarte curios.

Albert Einstein

Prima parte a cărții va prezenta, pe parcursul a opt capitole, caracteristicile esențiale ale limbajului Java. Nu vom include aici descrierea bibliotecilor suplimentare care nu sunt esențiale pentru înțelegerea limbajului, cum ar fi clasele grafice (AWT, Swing) sau *applet*-urile. De asemenea, tehnologii J2EE avansate, precum *JNI*, *RMI*, *JDBC*, nu sunt prezente nici ele, deoarece descrierea fiecăreia ar necesita cel puțin un volum separat.

După ce în primul capitol ați făcut cunoștință cu universul Java și v-ați familiarizat cu terminologia specifică, capitolul de față va începe prezentarea limbajului Java prin discutarea tipurilor primitive, a operațiilor elementare, a instrucțiunilor de decizie și iterative, a metodelor etc.

În acest capitol vom afla despre:

- Tipurile primitive ale limbajului Java, inclusiv operațiile care se pot realiza cu variabile de tipuri primitive;
- Cum sunt implementate instrucțiunile de decizie (condiționale) și cele iterative în Java;
- Noțiuni elementare despre metode și supraîncărcarea metodelor.

2.1 Mediul de lucru Java

Cum sunt introduse, compilate și rulate programele Java? Răspunsul depinde, evident, de platforma concretă pe care lucrați. Anexa A vă pune la dispoziție detalii referitoare la modul de instalare a mediului de lucru Java pentru diverse sisteme de operare.

Așa cum am arătat deja în primul capitol, codul sursă Java este conținut în fișiere text care au extensia `.java`. Compilatorul, care este de obicei `javac` sau `jikes`¹, compilează programul și generează fișiere `.class` care conțin *bytecode*. *Bytecode* este un limbaj intermediar *portabil* care este interpretat de către mașina virtuală Java, lansată prin comanda `java`. Detalii despre compilatoarele și interpretorul (mașina virtuală) Java, inclusiv modul de utilizare, se află tot în anexa A. Pentru programele Java, datele de intrare pot să provină din diverse surse:

- de la terminal, numit aici *standard input*;
- parametri suplimentari precizați la invocarea programului - parametri în linia de comandă;
- o componentă GUI (*Graphical User Interface*);
- un fișier.

2.2 Primul program Java

În această secțiune vom vedea Java în acțiune, prin crearea și rularea primei aplicații scrise în acest limbaj de programare. Să începem prin a examina programul simplu din **Listing 2.1**. Acest program tipărește un scurt mesaj pe ecran. Numerele din stânga fiecărei linii nu fac parte din program. Ele sunt furnizate doar pentru o mai ușoară referire a secvențelor de cod.

Listing 2.1: Un prim program simplu

```

1 //Primul program
2 public class FirstProgram
3 {
4     public static void main(String [] args)
5     {
6         System.out.println("Primul meu program Java");
    }

```

¹ `javac` este compilatorul realizat de Sun, `jikes` este compilatorul realizat de IBM și este preferat de mulți programatori deoarece este mult mai rapid.

```
7     }  
8 }
```

Transpuneți programul într-un fișier cu numele `FirstProgram.java`² după care compilați-l și rulați-l. Acest lucru se realizează astfel:

- executați comanda următoare în directorul în care este salvat fișierul sursă `FirstProgram.java` (folosiți un *command prompt* de tipul Windows MS-DOS Prompt):

```
javac FirstProgram.java
```

- dacă execuția comezii anterioare s-a realizat fără erori, executați comanda următoare în același director:

```
java FirstProgram
```

Rezultatul execuției programului este afișarea mesajului "Primul meu program Java". Este important de reținut că Java este *case-sensitive*, ceea ce înseamnă că face deosebiri între literele mari și mici.

2.2.1 Comentarii

În Java există trei tipuri de comentarii. Prima formă, care este moștenită de la limbajul C începe cu `/*` și se termină cu `*/`. Iată un exemplu:

```
1 /* Acesta este un comentariu  
2 pe doua linii */
```

Comentariile nu pot fi imbricate, deci nu putem avea un comentariu în interiorul altui comentariu.

Cea de-a doua formă de comentarii este moștenită de la limbajul C++ și începe cu `//`. Nu există simbol pentru încheiere, deoarece un astfel de comentariu se extinde automat până la sfârșitul liniei curente. Acest comentariu este folosit în linia 1 din **Listing 2.1**.

Cea de-a treia formă este asemănătoare cu prima doar că începe cu `/**` în loc de `/*`. Această formă de comentariu este utilizată pentru a furniza informații utilitarului `javadoc`, prezentat pe larg în anexa B.

Comentariile au fost introduse pentru a face codul mai lizibil pentru programatori. Un program bine comentat reprezintă un semn al unui bun programator.

²Atenție la literele mari și mici, deoarece compilatorul Java face distincție între majuscule și minuscule!

2.2.2 Funcția `main`

Un program Java constă dintr-o colecție de clase care interacționează între ele prin intermediul metodelor. Echivalentul Java al unei proceduri sau funcții din Pascal sau C este *metoda statică*, pe care o vom descrie puțin mai târziu în acest capitol. Atunci când se execută un program Java, mașina virtuală va căuta și invoca automat metoda statică având numele `main`. Linia 4 din **Listing 2.1** arată că metoda `main` poate fi opțional invocată cu anumiți parametri în linia de comandă. Tipul parametrilor funcției `main` cât și tipul funcției, `void`, sunt obligatorii.

2.2.3 Afișarea pe ecran

Programul din **Listing 2.1** constă dintr-o singură instrucțiune, aflată la linia 6. Funcția `println` reprezintă principalul mecanism de scriere în Java, fiind echivalent într-o anumită măsură cu funcția `writeln` din Pascal sau `printf` din C. În această situație se scrie un șir de caractere la fluxul de ieșire standard `System.out`. Vom discuta despre citire/scriere mai pe larg în cadrul capitolului 7. Deocamdată ne mulțumim doar să amintim că aceeași sintaxă este folosită pentru a scrie orice fel de entitate, fie că este vorba despre un întreg, real, șir de caractere sau alt tip.

2.3 Tipuri de date primitive

Java definește opt tipuri primitive de date, oferind în același timp o foarte mare flexibilitate în a defini noi tipuri de date, numite *clase*. Totuși, în Java există câteva diferențe esențiale între tipurile de date primitive și cele definite de utilizator. În această secțiune vom examina tipurile primitive și operațiile fundamentale care pot fi realizate asupra lor.

2.3.1 Tipurile primitive

Java are opt tipuri de date primitive prezentate în **Tabela 2.1**.

Cel mai des utilizat este tipul întreg specificat prin cuvântul cheie `int`. Spre deosebire de majoritatea altor limbaje de programare, marja de valori a tipurilor întregi nu este dependentă de mașină³. Java acceptă și tipurile întregi `byte`,

³Aceasta înseamnă că indiferent pe ce mașină rulăm programul nostru (fie că este un super-computer dotat cu 64 de procesoare, sau un simplu telefon mobil), variabilele de tip `int` vor fi reprezentate pe 32 de biți, lucru care nu este valabil pentru alte limbaje de programare cum ar fi C sau C++.

short și long, fiecare fiind adecvat pentru stocare de date întregi având anumite limite. Numerele reale (virgulă mobilă) sunt reprezentate în Java prin tipurile float și double. Tipul double are mai multe cifre semnificative, de aceea utilizarea lui este recomandată în locul tipului float. Tipul char este folosit pentru a reprezenta caractere. Un char ocupă în Java 16 biți (și nu doar 8, cum s-ar aștepta programatorii C sau Pascal, obișnuiți cu caractere ASCII) pentru a putea reprezenta toate caracterele din standardul *Unicode*. Acest standard conține peste 30.000 de caractere distincte care acoperă principalele limbi scrise (inclusiv limbile japoneză, chineză etc.). Prima parte a tabelului *Unicode* este identică cu tabela *ASCII*, deci setul de caractere Unicode este o extensie a mai vechiului set de caractere ASCII. Ultimul tip primitiv al limbajului Java este boolean; o variabilă de tip boolean poate lua una din valorile de adevăr true sau false.

Tabela 2.1: Cele 8 tipuri primitive de date în Java

Tip dată	Ce reține	Valori
byte	întreg pe 8 biți	-128 la 127
short	întreg pe 16 biți	-32768 la 32767
int	întreg pe 32 biți	-2.147.483.648 la 2.147.483.647
long	întreg pe 64 biți	-2^{63} la $2^{63} - 1$
float	virgulă mobilă pe 32 biți	6 cifre semnificative (10^{-46} la 10^{38})
double	virgulă mobilă pe 64 biți	15 cifre semnificative (10^{-324} la 10^{308})
char	caracter unicode	
boolean	variabilă booleană	false și true

2.3.2 Constante

Constantele întregi pot fi reprezentate în bazele 10, 8 sau 16. Notăția *octală* este indicată printr-un 0 ne semnificativ la început, iar notăția *hexa* este indicată printr-un 0x sau 0X la început. Iată reprezentarea întregului 37 în câteva moduri echivalente: 37, 045, 0x25. Notățiile octale și hexazecimale nu vor fi utilizate în această carte. Totuși trebuie să fim conștienți de ele pentru a folosi 0-uri la început doar acolo unde chiar vrem aceasta.

O *constantă caracter* este cuprinsă între apostrofuri, cum ar fi 'a'. Intern, Java interpretează această constantă ca pe un număr (codul *Unicode* al caracterului respectiv). Ulterior, funcțiile de scriere vor transforma acest număr în caracterul corespunzător. Constantele caracter mai pot fi reprezentate și în

forma:

```
'\uxxxx'.
```

unde xxxx este un număr în baza 16 reprezentând codul *Unicode* al caracterului.

Constantele de tip șir de caractere sunt cuprinse între ghilimele, ca în "Primul meu program Java". Există anumite secvențe speciale, numite *secvențe escape*, care sunt folosite pentru anumite caractere speciale. Noi vom folosi mai ales

```
'\n', '\\', '\'', '\"',
```

care înseamnă, respectiv, *linie nouă*, *backslash*, *apostrof* și *ghilimele*.

2.3.3 Declararea și inițializarea tipurilor primitive în Java

Orice variabilă Java, inclusiv cele primitive, este declarată prin descrierea numelui, a tipului și, opțional, a valorii inițiale. Numele variabilei trebuie să fie un *identificator*. Un identificator poate să conțină orice combinație de litere, cifre și caracterul *underscore* (liniuța de subliniere "_"). Identificatorii nu pot începe cu o cifră. Cuvintele rezervate, cum ar fi `int` nu pot fi identificatori. Nu pot fi utilizați nici identificatorii care deja sunt declarați și sunt vizibili.

Java este *case-sensitive*, ceea ce înseamnă că `sum` și `Sum` reprezintă identificatori diferiți. Pe parcursul acestei cărți vom folosi următoarea convenție pentru numele variabilelor:

- toate numele de variabilă încep cu literă mică, iar cuvintele noi din cadrul numelui încep cu literă mare. De exemplu: `sumaMaxima`, `nodVizitat`, `numberOfStudents` etc.;
- numele claselor începe cu literă mare. De exemplu: `ArithmeticException`, `FirstProgram`, `BinaryTree` etc.

Pentru detalii complete asupra convențiilor vă recomandăm anexa B, unde veți găsi informații despre convențiile de scriere a programelor Java, precum și despre o aplicație foarte utilă oferită de Sun, și anume `javadoc`, care permite foarte ușor crearea unor documentații ale programelor dumneavoastră.

Iată câteva exemple de declarații de variabile:

```
1 int numarElemente;
2 double mediaGenerală;
3 int produs = 1, suma = 0;
4 int produs1 = produs;
```

O variabilă este bine să fie declarată imediat înainte de a fi folosită. Așa cum vom vedea mai târziu, locul unde este declarată determină domeniul de vizibilitate și semnificația ei.

Limbajul Java permite și definirea de constante. Deoarece acestea necesită cunoștințe mai aprofundate, ne rezumăm la a spune că ele vor fi prezentate în cadrul subcapitolului destinat atributelor statice.

2.3.4 Citire/scriere la terminal

Scrierea la terminal în Java se realizează cu funcția `println` și nu pune probleme majore. Lucrurile nu stau deloc la fel cu citirea de la tastatură, care se realizează mult mai anevoios. Acest lucru se datorează în primul rând faptului că programele Java nu sunt concepute pentru a citi de la tastatură. În imensa majoritate a cazurilor, programele Java își preiau datele dintr-o *interfață grafică* (*Applet-urile*), din forme *HTML* (*Java Servlets*, *Java Server Pages*) sau din *fișiere*, fie că sunt fișiere standard (fișiere text, binare etc.) sau colecții de resurse (*resource bundles*).

Citirea și scrierea de la consolă sunt realizate prin metodele `readLine`, respectiv `println`. *Fluxul de intrare standard* este `System.in`, iar *fluxul de ieșire standard* este `System.out`.

Mecanismul de bază pentru citirea/scrierea formatată folosește tipul `String`, care va fi descris în capitolul următor. La afișare, operatorul `+` concatenează două `String`-uri. Pentru tipurile primitive, dacă parametrul scris nu este de tip `String` se face o conversie temporară la `String`. Aceste conversii pot fi definite și pentru obiecte, așa cum vom arăta mai târziu. Pentru citire se asociază un obiect de tipul `BufferedReader` cu `System.in`. Apoi se citește un `String` care va fi ulterior prelucrat. Capitolul 7 vă va edifica mai mult asupra acestor operații.

2.4 Operatori de bază

Această secțiune descrie operatorii de bază care sunt disponibili în Java. Acești operatori sunt utilizați pentru a crea expresii. O constantă sau o variabilă reprezintă o *expresie*, la fel ca și combinațiile de constante și variabile cu operatori. O expresie urmată de simbolul `;` reprezintă o *instrucțiune simplă*.

2.4.1 Operatori de atribuire

Programul simplu din **Listing 2.2** ilustrează câțiva operatori Java. *Operatorul de atribuire* este *semnul egal* (=). De exemplu, în linia 12, variabilei *a* i se atribuie valoarea variabilei *b* (care în acel moment are valoarea 7). Modificările ulterioare ale variabilei *b* nu vor afecta variabila *a*. Operatorii de atribuire pot fi înlanțuiți ca în următorul exemplu, atribuirile făcându-se (asociindu-se) de la dreapta la stânga:

```
z = y = x = 0;
```

Un alt operator de atribuire este += al cărui mod de utilizare este ilustrat în linia 18. Operatorul += adaugă valoarea aflată la dreapta (operatorului) la variabila din stânga. Astfel, valoarea lui *b* este incrementată de la 7 la 11. Java oferă și alți operatori de atribuire cum ar fi -=, *= și /= care modifică variabila aflată în partea stângă prin scădere, înmulțire și respectiv împărțire.

Listing 2.2: Program care ilustrează anumiți operatori simpli

```
1 /** Utilizarea operatorilor. */
2 public class OperatorTest
3 {
4     public static void main(String[] args)
5     {
6         int a = 5, b = 7, c = 4;
7
8         System.out.println("a=" + a); //a=5
9         System.out.println("b=" + b); //b=7
10        System.out.println("c=" + c); //c=4
11
12        a = b;
13
14        System.out.println("a=" + a); //a=7
15        System.out.println("b=" + b); //b=7
16        System.out.println("c=" + c); //c=4
17
18        b += c;
19
20        System.out.println("a=" + a); //a=7
21        System.out.println("b=" + b); //b=11
22        System.out.println("c=" + c); //c=4
23
24        c = a + b;
25
26        System.out.println("a=" + a); //a=7
27        System.out.println("b=" + b); //b=11
28        System.out.println("c=" + c); //c=18
29
30        c++;
```

```

31
32     System.out.println("a=" + a); //a=7
33     System.out.println("b=" + b); //b=11
34     System.out.println("c=" + c); //c=19
35
36     ++a;
37
38     System.out.println("a=" + a); //a=8
39     System.out.println("b=" + b); //b=11
40     System.out.println("c=" + c); //c=19
41
42     b = ++a + c++;
43
44     System.out.println("a=" + a); //a=9
45     System.out.println("b=" + b); //b=28
46     System.out.println("c=" + c); //c=20
47 }
48 }

```

2.4.2 Operatori aritmetici binari

Linia 24 din **Listing 2.2** ilustrează unul dintre operatorii binari tipici ai limbajelor de programare: *operatorul de adunare* (+). Operatorul + are ca efect adunarea conținutului variabilelor a și b; valorile lui a și b rămân neschimbate. Valoarea rezultată este atribuită lui c. Alți operatori aritmetici folosiți în Java sunt: -, *, / și % utilizați respectiv pentru *scădere*, *înmulțire*, *împărțire* și *rest*.

Împărțirea a două valori întregi are ca valoare doar partea întreagă a rezultatului. De exemplu $3/2$ are valoarea 1, dar $3.0/2$ are valoarea 1.5, deoarece unul dintre operanzi (3.0) a fost o valoare în virgulă mobilă, și astfel rezultatul întregii expresii este convertit la virgulă mobilă.

Așa cum este și normal, adunarea și scăderea au aceeași prioritate. Această prioritate este mai mică decât cea a grupului format din înmulțire, împărțire și rest; astfel $1 + 2 * 3$ are valoarea 7. Toți acești operatori sunt evaluați de la stânga la dreapta (astfel $3 - 2 - 2$ are valoarea -1). Toți operatorii aritmetici au o anumită prioritate și o anumită asociere (fie de la stânga la dreapta, fie de la dreapta la stânga, descrisă în **Tabela 2.2⁴**).

2.4.3 Operatori aritmetici unari

În plus față de operatorii aritmetici binari care necesită doi operanzi, Java dispune și de *operatori unari* care necesită doar un singur operand. Cel mai

⁴Prin asociere înțelegem ordinea de evaluare într-o expresie care conține operatori de același tip și nu are paranteze (de exemplu, $a - b - c$).

cunoscut operator unar este *operatorul minus* ($-$) care returnează operandul cu semn opus. Astfel, $-x$ este opusul lui x .

Java oferă de asemenea *operatorul de autoincrementare* care adaugă 1 la valoarea unei variabile, notat prin $++$, și *operatorul de autodecrementare* care scade 1 din valoarea variabilei, notat cu $--$. Un caz banal de utilizare a acestor operatori este exemplificat în liniile 30 și 36 din **Listing 2.2**. În ambele cazuri operatorul $++$ adaugă 1 la valoarea variabilei. În Java, ca și în C, orice expresie are o valoare. Astfel, un operator aplicat unei variabile generează o expresie cu o anumită valoare. Deși faptul că variabila este incrementată înainte de execuția următoarei instrucțiuni este garantat, se pune întrebarea: "Care este valoarea expresiei de autoincrementare dacă ea este utilizată în cadrul unei alte expresii?"

În acest caz, locul unde se plasează operatorul $++$ este esențial. Semnificația lui $++x$ este că valoarea expresiei este egală cu noua valoare a lui x . Acest operator este numit *incrementare prefixată*. În mod analog, $x++$ înseamnă că valoarea expresiei este egală cu valoarea originală a lui x . Acesta este numit *incrementare postfixată*. Aceste trăsături sunt exemplificate în linia 42 din **Listing 2.2**. Atât a , cât și c sunt incrementate cu 1, iar b este obținut prin adunarea valorii incrementate a lui a (care este 9) cu valoarea inițială a lui c (care este 19).

2.4.4 Conversii de tip

Operatorul conversie de tip, numit în jargonul programatorilor și *operatorul de cast*, este utilizat pentru a genera o variabilă temporară de un nou tip. Să considerăm, de exemplu, secvența de cod:

```
1 double rest;
2 int x = 2;
3 int y = 5;
4 rest = x / y; //probabil incorect!
```

La efectuarea operației de împărțire, atât x cât și y fiind numere întregi, se va realiza o împărțire întreagă și se obține 0. Întregul 0 este apoi convertit implicit la *double* astfel încât să poată fi atribuit lui *rest*. Probabil că intenția noastră era aceea de a atribui lui *rest* valoarea 0.4. Soluția este de a converti temporar pe x sau pe y la *double*, pentru ca împărțirea să se realizeze în virgulă mobilă. Acest lucru se poate obține astfel:

```
rest = (double) x / y;
```

De remarcat că nici x și nici y nu se schimbă. Se crează o variabilă temporară fără nume, având valoarea 2.0, iar valoarea ei este utilizată pentru a efectua

împărțirea. Operatorul de conversie de tip are o prioritate mai mare decât operatorul de împărțire, de aceea conversia de tip se efectuează înainte de a se efectua împărțirea.

2.5 Instrucțiuni condiționale

Această secțiune este dedicată instrucțiunilor care controlează fluxul de execuție al programului: instrucțiunile condiționale și iterația.

2.5.1 Operatori relaționali

Testul fundamental care poate fi realizat asupra tipurilor primitive este *comparația*. Comparația se realizează utilizând *operatorii de egalitate/inegalitate* și *operatorii de comparație* (<, > etc.). În Java, operatorii de egalitate/inegalitate sunt == respectiv !=. De exemplu,

```
exprStanga == exprDreapta
```

are valoarea true dacă exprStanga și exprDreapta sunt egale, altfel are valoarea false. Analog, expresia:

```
exprStanga != exprDreapta
```

are valoarea true dacă exprStanga și exprDreapta sunt diferite; altfel are valoarea false.

Operatorii de comparație sunt <, <=, >, >= iar semnificația lor este cea naturală pentru tipurile fundamentale. Operatorii de comparație au prioritate mai mare decât operatorii de egalitate. Totuși, ambele categorii au prioritate mai mică decât operatorii aritmetici, dar mai mare decât operatorii de atribuire. Astfel, veți constata că în cele mai multe cazuri folosirea parantezelor nu va fi necesară. Toți acești operatori se evaluează de la stânga la dreapta, dar cunoașterea acestui lucru nu ne folosește prea mult. De exemplu, în expresia `a < b < 6`, prima comparație generează o valoare booleană, iar a doua expresie este greșită, deoarece operatorul < nu este definit pentru valori booleene. Paragraful următor descrie cum se poate realiza acest test în mod corect.

2.5.2 Operatori logici

Java dispune de *operatori logici* care sunt utilizați pentru a simula operatorii *and*, *or* și *not* din algebra *Booleană*. Acești operatori sunt referiți uneori și sub numele de *conjunție*, *disjuncție* și, respectiv, *negare*, simbolurile corespunzătoare fiind &&, || și !. Implementarea corectă a testului din paragraful anterior este:

(a < b) && (b < 6)

Prioritatea conjuncției și a disjuncției este suficient de mică față de prioritatea celorlalți operatori din expresie pentru ca parantezele să nu fie necesare. && are prioritate mai mare decât | |, iar ! are aceeași prioritate cu alți operatori unari (++ , -- , vezi **Tabela 2.2**).

Tabela 2.2: Operatori Java listați în ordinea priorității

Categorie	Exemple	Asociere
Operatori pe referințe	. []	Stânga la dreapta
Unari	++ -- ! - (tip)	Dreapta la stânga
Multiplcativi	* / %	Stânga la dreapta
Aditivi	+ -	Stânga la dreapta
Shiftare (pe biți)	<< >> >>>	Stânga la dreapta
Relaționali	< <= > >= instanceof	Stânga la dreapta
Egalitate	== !=	Stânga la dreapta
AND pe biți	&	Stânga la dreapta
XOR pe biți	^	Stânga la dreapta
OR pe biți		Stânga la dreapta
AND logic	&&	Stânga la dreapta
OR logic		Stânga la dreapta
Condițional	?:	Dreapta la stânga
Atribuire	= *= /= %= += -=	Dreapta la stânga

O regulă importantă este că operatorii && și | | folosesc evaluarea booleană scurtcircuitată⁵. Aceasta înseamnă că dacă rezultatul poate fi determinat evaluând prima expresie, a doua expresie nu mai este evaluată. De exemplu, în expresia:

```
x != 0 && 1/x != 3
```

dacă x este 0, atunci prima jumătate este false. Aceasta înseamnă că rezultatul conjuncției va fi fals, deci a doua expresie nu mai este evaluată. Acesta este un detaliu important, deoarece împărțirea la 0 ar fi generat un comportament eronat.

⁵Numită uneori și evaluare booleană parțială.

2.5.3 Operatori la nivel de bit

Operatorii la nivel de bit pe care limbajul Java îi pune la dispoziția programatorului se împart în două grupe:

- operatori de deplasare (*shiftare*) : `>>` `<<` `>>>`
- operatori logici : `&` `|` `^` `~`

Fie că fac parte din prima sau a doua categorie, operatorii la nivel de bit nu pot fi utilizați decât în cazul numerelor întregi (variabile de tipul `byte`, `short`, `int` și `long`). De fapt, operatorii se aplică reprezentării binare a numerelor implicate. Cu alte cuvinte, dacă avem operația `5 & 3`, aceasta înseamnă că de fapt operatorul `&` se aplică reprezentării în baza 2 a numerelor 5 și 3, adică 101 și 11.

Prima grupă de operatori la nivel de bit, cea a operatorilor de deplasare, execută manipularea biților primului operand, deplasându-i la stânga sau la dreapta, în funcție de tipul operatorului (vezi **Tabela 2.3**).

Tabela 2.3: Operatori la nivel de bit pentru deplasare

Operator	Utilizare	Descriere
<code>>></code>	<code>op1 >> op2</code>	deplasarea biților lui <code>op1</code> la dreapta cu <code>op2</code> poziții
<code><<</code>	<code>op1 << op2</code>	deplasarea biților lui <code>op1</code> la stânga cu <code>op2</code> poziții
<code>>>></code>	<code>op1 >>> op2</code>	deplasarea biților lui <code>op1</code> la dreapta cu <code>op2</code> poziții, <code>op1</code> fiind considerat <code>unsigned</code>

Fiecare operator deplasează biții operandului din stânga cu un număr de poziții indicat de operandul din dreapta. Deplasarea se face în sensul indicat de operator (`>>` la dreapta, iar `<<` la stânga). De exemplu, următoarea instrucțiune deplasează biții numărului 5 la dreapta cu o poziție:

```
5 >> 1;
```

Reprezentarea binară a numărului 5 este 101. Prin deplasarea biților la dreapta cu o poziție se obține numărul care are reprezentarea binară 10, deoarece primul 1 (cel din stânga) trece cu o poziție mai la dreapta, deci în locul lui 0, care trece cu o poziție mai la dreapta, deci în locul celei de-a doua cifre 1,

care se pierde. Biții din stânga care au rămas descompletați se completează cu 0, ceea ce conduce la rezultatul final 010. Astfel, am obținut numărul 2 în reprezentarea zecimală.

În concluzie, $5 \gg 1 = 2$. După cum se poate observa, pentru numerele pozitive, deplasarea la dreapta cu $\text{op}2$ biți este echivalentă cu împărțirea lui $\text{op}1$ la $2^{\text{op}2}$. Pentru exemplul nostru, $5/2^1 = 2$.

Analog se întâmplă și în cazul deplasării la stânga. Instrucțiunea următoare deplasează biții numărului 5, la stânga, cu o poziție:

$5 \ll 1;$

Rezultatul deplasării biților 101 la stânga cu o poziție este 1010, adică numărul 10 în reprezentare zecimală. Acest lucru este echivalent cu înmulțirea cu $2^{\text{op}2}$. În cazul nostru, $5 * 2^1 = 10$.

Cel de-al treilea tip de deplasare a biților, \ggg , este asemănător cu \gg , cu specificarea că numărul $\text{op}1$ este considerat fără semn (unsigned).

Cea de-a doua categorie de operatori la nivel de bit realizează operații logice ȘI, SAU, SAU EXCLUSIV (XOR) și NEGARE:

Tabela 2.4: Operatori pentru operații logice la nivel de bit

Operator	Utilizare	Descriere
&	$\text{op}1 \ \& \ \text{op}2$	ȘI la nivel de biți
	$\text{op}1 \ \ \text{op}2$	SAU la nivel de biți
^	$\text{op}1 \ ^ \ \text{op}2$	SAU EXCLUSIV la nivel de biți
~	$\sim \text{op}1$	NEGARE la nivel de biți

Să presupunem că avem două numere, 5 și 3, cărora dorim să le aplicăm operații logice la nivel de biți. Operațiile logice la nivel de biți constau în aplicarea operației respective perechilor de biți de pe poziții egale în cele două numere (cu excepția operației de negare care este unară). În situația în care numerele nu au reprezentarea binară de aceeași lungime, reprezentarea mai scurtă este completată cu zerouri ne semnificative (inserate în fața reprezentării), până se obțin dimensiuni egale.

Prezentate pe scurt, operațiile logice pe perechi de biți se realizează pe baza următorului alogritm:

- expresia $b1 \ \& \ b2$ are valoarea 1, dacă și numai dacă $b1$ și $b2$ au valoarea 1 ($b1$ și $b2$ sunt 2 biți, care pot lua, evident, doar valorile 0 și 1). Altfel, expresia are valoarea 0;
- expresia $b1 \ | \ b2$ are valoarea 0, dacă și numai dacă $b1$ și $b2$ au valoarea 0. Altfel, expresia are valoarea 1;
- expresia $b1 \ ^ \ b2$ are valoarea 1, dacă și numai dacă unul dintre operanzi este 0, iar celălalt 1. Altfel, expresia are valoarea 0;
- expresia $\sim b1$ reprezintă negarea lui $b1$, deci are valoarea 1, dacă $b1$ este 0, sau are valoarea 0, dacă $b1$ este 1.

Iată câteva exemple de utilizare:

- $5 \ \& \ 3 = 1$

```

      101    // = 5
&   011    // = 3
-----
      001    // = 1

```

- $5 \ | \ 3 = 7$

```

      101    // = 5
|   011    // = 3
-----
      111    // = 7

```

- $5 \ ^ \ 3 = 6$

```

      101    // = 5
^   011    // = 3
-----
      110    // = 6

```

- $\sim 5 = -6$

```

~  0...0101    // = 5
-----
  1...1010    // = -6

```

Această operație necesită anumite precizări. După cum am văzut anterior, un număr de tip `int` este reprezentat pe 32 de biți. Cu alte cuvinte, reprezentarea numărului 5 în baza 2, are de fapt 32 de cifre binare, dintre care 29 sunt zerouri nesemnificative (cele din față), pe care le-am omis până acum din motive de spațiu. La o operație de negare, toate zerourile nesemnificative ale reprezentării binare devin 1 și capătă astfel importanță (primul bit este bitul de semn, care se modifică și el). Pentru a înțelege mai bine modul în care se reprezintă numerele întregi în memorie vă recomandăm lucrarea [Boian].

2.5.4 Instrucțiunea `if`

Instrucțiunea `if` este instrucțiunea fundamentală de *decizie*. Forma sa simplă este:

```
1 if ( expresie )
2     instructiune
3
4 urmatoareaInstructiune
```

Dacă `expresie` are valoarea `true` atunci se execută `instructiune`; în caz contrar, `instructiune` nu se execută. După ce instrucțiunea `if` se încheie fără incidente (vezi cazul excepțiilor netratate, în capitolul 6), controlul este preluat de `urmatoareaInstructiune`.

Opțional, putem folosi instrucțiunea `if-else` după cum urmează:

```
1 if ( expresie )
2     instructiune1
3 else
4     instructiune2
5
6 urmatoareaInstructiune
```

În acest caz, dacă `expresie` are valoarea `true`, atunci se execută `instructiune1`; altfel se execută `instructiune2`. În ambele cazuri controlul este apoi preluat de `urmatoareaInstructiune`. Iată un exemplu:

```
1 System.out.println("1/x este: ");
2 if ( x != 0 )
3     System.out.print( 1/x );
4 else
5     System.out.print("Nedefinit");
6
7 System.out.println();
```

De reținut că doar o *singură* instrucțiune poate exista pe ramura de `if` sau de `else` indiferent de cum indentați codul. Iată două erori frecvente pentru

începători:

```
1 if (x == 0) ; //instrucțiune vida!!!
2   System.out.println("x este 0");
3 else
4   System.out.print("x este");
5   System.out.println(x); //instrucțiune in afara clauzei else
```

Prima greșeală constă în a pune ; după if. Simbolul ; reprezintă în sine *instrucțiunea vidă*; ca o consecință, acest fragment de cod nu va fi compilabil (else nu va fi asociat cu nici un if). După ce am corectat această eroare, rămânem cu o eroare de logică: ultima linie de cod nu face parte din if, deși acest lucru este sugerat de indentare. Pentru a rezolva această problemă vom utiliza un bloc în care grupăm o secvență de instrucțiuni printr-o pereche de acolade:

```
1 if (x == 0)
2 {
3   System.out.println("x este 0");
4 }
5 else
6 {
7   System.out.print("x este");
8   System.out.println(x);
9 }
```

Practic, prin cuprinderea mai multor instrucțiuni între acolade, creăm o singură instrucțiune, numită *instrucțiune compusă*. Așadar, instrucțiune1 (ca și instrucțiune2) poate fi o instrucțiune simplă sau o instrucțiune compusă. Instrucțiunile compuse sunt formate dintr-o succesiune de instrucțiuni (simple sau compuse) cuprinse între acolade. Această definiție permite imbricarea instrucțiunilor compuse în cadrul altor instrucțiuni compuse. În exemplul anterior, pe ramura if avem o instrucțiune compusă formată doar dintr-o singură instrucțiune simplă:

```
1 {
2   System.out.println("x este 0");
3 }
```

Am folosit aici o instrucțiune compusă deși, fiind vorba de o singură instrucțiune simplă, acest lucru nu era absolut necesar. Totuși, această practică îmbunătățește foarte mult lizibilitatea codului și vă invităm și pe dumneavoastră să o adoptați.

Instrucțiunea if poate să facă parte dintr-o altă instrucțiune if sau else, la fel ca și celelalte instrucțiuni de control prezentate în continuare în această secțiune.

2.5.5 Instrucțiunea `while`

Java, ca și Pascal sau C, dispune de trei *instrucțiuni de ciclare* (repetitive): instrucțiunea *while*, instrucțiunea *do* și instrucțiunea *for*. Sintaxa instrucțiunii *while* este:

```
1 while ( expresie )
2     instructiune
3
4 urmatoareaInstructiune
```

Observați că, la fel ca și la instrucțiunea *if*, nu există `;` în sintaxă. Dacă apare un `;` după *while*, va fi considerat ca instrucțiune vidă.

Cât timp *expresie* este *true* se execută *instructiune*; apoi *expresie* este evaluată din nou. Dacă *expresie* este *false* de la bun început, atunci *instructiune* nu va fi executată niciodată. În general, *instructiune* face o acțiune care ar putea modifica valoarea lui *expresie*; altfel, ciclarea s-ar putea produce la infinit. Când execuția instrucțiunii *while* se încheie, controlul este preluat de *urmatoareaInstructiune*.

2.5.6 Instrucțiunea `for`

Instrucțiunea *while* ar fi suficientă pentru a exprima orice fel de ciclare. Totuși, Java mai oferă încă două forme de a realiza ciclarea: instrucțiunea *for* și instrucțiunea *do*. Instrucțiunea *for* este utilizată în primul rând pentru a realiza iterația. Sintaxa ei este:

```
1 for ( initializare ; test ; actualizare )
2     instructiune
3
4 urmatoareaInstructiune
```

În această situație, *initializare*, *test* și *actualizare* sunt toate expresii, și toate trei sunt opționale. Dacă *test* lipsește, valoarea sa implicită este *true*. După paranteza de închidere nu se pune `;`.

Instrucțiunea *for* se execută realizând mai întâi *initializare*. Apoi, cât timp *test* este *true* se execută *instructiune*, iar apoi se execută *actualizare*. Dacă *initializare* și *actualizare* sunt omise, instrucțiunea *for* se va comporta exact ca și instrucțiunea *while*.

Avantajul instrucțiunii *for* constă în faptul că se poate vedea clar marja pe care iterează variabilele contor.

Următoarea secvență de cod afișează primele 100 numere întregi pozitive:

```
1 for ( int i = 1; i < 100; ++i )
2 {
```

```
3     System.out.println(i);
4 }
```

Acest fragment ilustrează și practica obișnuită pentru programatorii Java (și C++) de a declara un contor întreg în secvența de inițializare a ciclului. Durata de viață a acestui contor se extinde doar în interiorul ciclului.

Atât inițializare cât și actualizare pot folosi operatorul *virgulă* pentru a permite *expresii multiple*. Următorul fragment ilustrează această tehnică frecvent folosită:

```
1 for (i = 0, sum = 0; i <= n; i++, sum += n)
2 {
3     System.out.println(i + "\t" + sum);
4 }
```

Ciclurile pot fi imbricate la fel ca și instrucțiunile `if`. De exemplu, putem găsi toate perechile de numere mici a căror sumă este egală cu produsul lor (cum ar fi 2 și 2, a căror sumă și produs este 4) folosind secvența de cod de mai jos:

```
1 for (int i = 1; i <= 10; i++)
2 {
3     for (int j = 1; j <= 10; j++)
4     {
5         if (i + j == i * j)
6         {
7             System.out.println(i + ", " + j);
8         }
9     }
10 }
```

2.5.7 Instrucțiunea `do`

Instrucțiunea `while` realizează un test repetat. Dacă testul este `true` atunci se execută instrucțiunea din cadrul ei. Totuși, dacă testul inițial este `false`, instrucțiunea din cadrul ciclului nu este executată niciodată. În anumite situații avem nevoie ca instrucțiunile din ciclu să se execute cel puțin o dată. Acest lucru se poate realiza utilizând instrucțiunea `do`. Instrucțiunea `do` este asemănătoare cu instrucțiunea `while`, cu deosebirea că testul este realizat după ce instrucțiunile din corpul ciclului se execută. Sintaxa sa este:

```
1 do
2     instructiune
3 while (expresie);
4
5 urmatoareaInstructiune;
```


Remarcați faptul că instrucțiunea `do` se termină cu `;`. Un exemplu tipic în care se utilizează instrucțiunea `do` este dat de fragmentul de (pseudo-) cod de mai jos:

```
1 do
2 {
3     afiseaza mesaj;
4     citeste data;
5 }
6 while (data nu este corecta);
```

Instrucțiunea `do` este instrucțiunea de ciclare cel mai puțin utilizată. Totuși, când vrem să executăm cel puțin o dată instrucțiunile din cadrul ciclului, și `for` este incomod de utilizat, atunci `do` este alegerea potrivită.

2.5.8 Instrucțiunile `break` și `continue`

Instrucțiunile `for` și `while` au condiția de terminare *înaintea* instrucțiunilor care se repetă. Instrucțiunea `do` are condiția de terminare *după* instrucțiunile care se repetă. Totuși, în anumite situații, s-ar putea să fie nevoie să întrerupem ciclul chiar în mijlocul instrucțiunilor care se repetă. În acest scop, se poate folosi instrucțiunea `break`. De obicei, instrucțiunea `break` apare în cadrul unei instrucțiuni `if`, ca în exemplul de mai jos:

```
1 while (...)
2 {
3     ...
4     if (conditie)
5     {
6         break;
7     }
8     ...
9 }
```

În cazul în care sunt două cicluri imbricate, instrucțiunea `break` părăsește doar ciclul cel mai din interior. Dacă există mai mult de un ciclu care trebuie terminat, `break` nu va funcționa corect, și mai mult ca sigur că ați proiectat prost algoritmul. Totuși, Java oferă așa numitul `break` etichetat. În acest caz, o anumită instrucțiune de ciclare este etichetată și instrucțiunea `break` poate fi aplicată acelei instrucțiuni de ciclare, indiferent de numărul de cicluri imbricate. Iată un exemplu:

```
1 eticheta:
2     while (...)
3     {
4         while (...)
5         {
```

```

6          ...
7          if ( conditie )
8          {
9              break eticheta;
10         }
11     }
12 }
13 //controlul programului trece aici dupa executia lui break

```

În anumite situații dorim să renunțăm la execuția iterației curente din ciclu și să trecem la următoarea iterație a ciclului. Acest lucru poate fi realizat cu instrucțiunea `continue`. Ca și `break`, instrucțiunea `continue` este urmată de `;` și se aplică doar ciclului cel mai interior în cazul ciclurilor imbricate. Următorul fragment tipărește primele 100 de numere întregi, cu excepția celor divizibile cu 10:

```

1 for ( int i = 1; i <= 100; i++)
2 {
3     if ( i % 10 == 0 )
4     {
5         continue;
6     }
7
8     System.out.println(i);
9 }

```

Desigur, că exemplul de mai sus poate fi implementat și utilizând un `if` simplu. Totuși instrucțiunea `continue` este adeseori folosită pentru a evita imbricări complicate de tip `if-else` în cadrul ciclurilor.

2.5.9 Instrucțiunea `switch`

Instrucțiunea `switch` este numită uneori și *instrucțiune de selecție*; ea are rolul de a selecta dintre mai multe secvențe de cod, una care va fi executată, funcție de valoarea unei expresii întregi. Forma sa este:

```

1 switch ( expresie—selectare )
2 {
3     case valoare—intreaga1 :
4         instructiune;
5         break;
6     case valoare—intreaga2 :
7         instructiune;
8         break;
9
10    // ...
11
12    case valoare—intreagaN :

```

```

13         instructiune ;
14         break ;
15
16     default :
17         instructiune ;
18 }

```

expresie-selectare este o expresie care produce o valoare întreagă. Instrucțiunea `switch` compară valoarea expresiei `expresie-selectare` cu fiecare valoare-întreaga. Dacă are loc egalitatea, se execută instrucțiunea corespunzătoare (simplă sau compusă). Dacă nu are loc nici o egalitate se execută instrucțiunea din `default` (alternativa `default` este opțională în cadrul `switch`).

Observați că în exemplul de mai sus, fiecare `case` se încheie cu un `break` care are ca efect saltul la sfârșitul instrucțiunii `switch`. Acesta este modul obișnuit de a scrie o instrucțiune de tip `switch`, dar prezența instrucțiunii `break` nu este obligatorie. Dacă instrucțiunea `break` lipsește, atunci se va executa și codul corespunzător instrucțiunilor `case` următoare până când se întâlnește un `break`. Deși de obicei nu ne dorim un astfel de comportament, el poate fi uneori util pentru programatorii experimentați.

Pentru a exemplifica instrucțiunea `switch`, programul din **Listing 2.3** crează litere aleator și determină dacă acestea sunt vocale sau consoane (în limba engleză).

Listing 2.3: Program care exemplifică instrucțiunea `switch`

```

1 //VowelsAndConsonants.java
2 // Program demonstrativ pentru instructiunea switch
3 public class VowelsAndConsonants
4 {
5     public static void main(String[] args)
6     {
7         for (int i = 0; i < 100; i++)
8         {
9             char c = (char) (Math.random() * 26 + 'a');
10            System.out.print(c + ": ");
11            switch (c)
12            {
13                case 'a':
14                case 'e':
15                case 'i':
16                case 'o':
17                case 'u':
18                    System.out.println("vocala");
19                    break;
20                case 'y':
21                case 'w':

```

```

22         System.out.println("Uneori vocale ");
23         //doar in limba engleza!
24         break;
25     default:
26         System.out.println("consoana");
27     } //switch
28 } //for
29 } //main
30 } //class

```

Funcția `Math.random()` generează o valoare în intervalul $[0,1)$. Prin înmulțirea valorii returnate de această funcție cu numărul de litere din alfabet (26 litere) se obține un număr în intervalul $[0,26)$. Adunarea cu prima literă ('a', care are de fapt valoarea 97, codul ASCII al literei 'a') are ca efect transpunerea în intervalul $[97,123)$. În final se folosește operatorul de conversie de tip pentru a trunchia numărul la o valoare din mulțimea 97, 98, ..., 122, adică un cod ASCII al unui caracter din alfabetul englez.

2.5.10 Operatorul condițional

Operatorul condițional este folosit ca o prescurtare pentru instrucțiuni simple de tipul `if-else`. Forma sa generală este:

```
exprTest ? expresieDa : expresieNu;
```

Mai întâi se evaluează `exprTest` urmată fie de evaluarea lui `expresieDa` fie de cea a lui `expresieNu`, rezultând astfel valoarea întregii expresii. `expresieDa` este evaluată dacă `exprTest` are valoarea `true`; în caz contrar se evaluează `expresieNu`. Prioritatea operatorului condițional este chiar deasupra operatorilor de atribuire. Acest lucru permite omiterea parantezelor atunci când asignăm rezultatul operatorului condițional unei variabile. Ca un exemplu, minimul a două variabile poate fi calculat după cum urmează:

```
valMin = x < y ? x : y;
```

2.6 Metode

Ceea ce în alte limbaje de programare numeam procedură sau funcție, în Java este numit *metodă*. O definiție completă a noțiunii de metodă o vom da mai târziu, când vom introduce noțiunile de clasă și obiect. În acest paragraf prezentăm doar câteva noțiuni elementare pentru a putea scrie funcții de genul celor din C sau Pascal pe care să le folosim în câteva programe simple.

Listing 2.4: Declararea și apelul unei metode

```

1 public class Minim
2 {
3     public static void main( String [] args )
4     {
5         int a = 3 ;
6         int b = 7 ;
7         System.out.println( "Minimul este: " + min(a,b) ) ;
8     }
9
10    //declaratia metodei min
11    public static int min( int x, int y )
12    {
13        return x<y?x:y ;
14    }
15 }

```

Antetul unei metode constă dintr-un *nume*, o *listă* (eventual vidă) de *parametri* și un *tip* pentru valoarea returnată. Codul efectiv al metodei, numit adeseori corpul metodei, este format dintr-o instrucțiune compusă (o secvență de instrucțiuni cuprinsă între acolade). Definirea unei metode constă în *antet* și *corp*. Un exemplu de definire și utilizare a unei metode este dat în programul din **Listing 2.4**.

Prin prefixarea metodelor cu ajutorul cuvintelor cheie `public static` putem mima într-o oarecare măsură funcțiile din Pascal și C. Deși această tehnică este utilă în anumite situații, ea nu trebuie utilizată în mod abuziv.

Numele metodei este un identificator. *Lista de parametri* constă din 0 sau mai mulți *parametri formali*, fiecare având un tip precizat. Când o metodă este apelată, *parametrii actuali* sunt trecuți în parametrii formali utilizând atribuirea obișnuită. Aceasta înseamnă că tipurile primitive sunt transmise utilizând exclusiv transmiterea prin valoare. Parametrii actuali nu vor putea fi modificați de către funcție. Definirile metodelor pot apărea în orice ordine.

Instrucțiunea `return` este utilizată pentru a întoarce o valoare către codul apelant. Dacă tipul funcției este `void` atunci nu se întoarce nici o valoare. În anumite situații, pentru ieșirea forțată dintr-o metodă care are tipul `void` se folosește `return`; fără nici un parametru.

2.6.1 Supraîncărcarea numelor la metode

Să presupunem că dorim să scriem o metodă care calculează maximumul a trei numere întregi. Un antet pentru această metodă ar fi:

```
int max(int a, int b, int c)
```

În unele limbaje de programare (Pascal, C), acest lucru nu ar fi permis dacă există deja o funcție `max` cu doi parametri. De exemplu, se poate să avem deja declarată o metodă `max` cu antetul:

```
int max(int a, int b)
```

Java permite *supraîncărcarea* (engl. *overloading*) numelui metodelor. Aceasta înseamnă că mai multe metode cu același nume pot fi declarate în cadrul aceleiași clase atâta timp cât *semnăturile* lor (adică lista de parametri) diferă. Atunci când se face un apel al metodei `max`, compilatorul poate ușor să deducă despre care metodă este vorba examinând *lista parametrilor de apel*. Se poate să existe metode supraîncărcate cu același număr de parametri formali, atâta timp cât cel puțin unul din tipurile din lista de parametri este diferit.

De reținut faptul că tipul funcției nu face parte din semnătura ei. Aceasta înseamnă că nu putem avea două metode în cadrul aceleiași clase care să difere doar prin tipul valorii returnate. Metode din clase diferite pot avea același nume, parametri și chiar tip returnat, dar despre aceasta vom discuta pe larg mai târziu.

Rezumat

În acest capitol am discutat despre noțiunile fundamentale ale limbajului Java, cum ar fi tipurile primitive, operatorii, instrucțiunile conditionale și repetitive, precum și metode, care se regăsesc în aproape orice limbaj de programare.

Totuși, orice program nebanal presupune utilizarea tipurilor neprimitive, numite *tipuri referință*, care vor fi discutate în capitolul următor.

Noțiuni fundamentale

break: instrucțiune prin care se iese din cel mai din interior ciclu iterativ sau din cadrul instrucțiunii `switch`.

comentarii: au rolul de a face codul mai lizibil pentru programatori, dar nu au nici o valoare semantică. Java dispune de trei tipuri de comentarii.

continue: instrucțiune prin care se trece la următoarea iterație din cadrul ciclului iterativ în care este folosită.

do: instrucțiune repetitivă în care ciclul repetitiv se execută cel puțin o dată.

for: instrucțiune repetitivă utilizată mai ales pentru iterație.

identificator: nume dat unei variabile sau unei metode.

if: instrucțiune de decizie.

instrucțiune compusă: mai multe instrucțiuni cuprinse între acolade și care formează astfel o singură instrucțiune.

main: metodă specială în cadrul unei clase, care este apelată la execuția aplicației.

metodă: echivalentul Java al unei funcții.

metodă de tip static: metodă echivalentă cu o metodă globală.

operator condițional (? :): operator folosit ca o prescurtare a secvențelor simple `if...else`.

operatori aritmetici binari: utilizați pentru operații aritmetice de bază, cum ar fi `+`, `-`, `*`, `/` și `%`.

operatori de atribuire: operatori utilizați pentru modificarea valorii unei variabile. Exemple de operatori de atribuire sunt `=`, `+=`, `-=`, `*=`, `/=`.

operatori de egalitate: sunt `==` și `!=`. Ei întorc fie `true` fie `false`.

operatori de incrementare și decrementare: operatori care adaugă, respectiv scad, o unitate la/din valoarea curentă a variabilei. Există două forme incrementare și decrementare: prefixat și postfixat.

operatori logici: `&&`, `||` și `!`, folosiți pentru a reprezenta conceptele de disjuncție, conjuncție și negare din algebra booleană.

operatori relaționali: `<`, `>`, `<=`, `>=` sunt folosiți pentru a decide care din două valori este mai mică sau mai mare; rezultatul lor este `true` sau `false`.

return: instrucțiune utilizată pentru a returna informații către instrucțiunea care a apelat metoda.

semnătura unei metode: este formată din combinația dintre numele metodei și tipurile din lista de parametri. Tipul valorii returnate nu face parte din semnătură.

switch: instrucțiune `if` generalizată.

tipuri întregi: `byte`, `short`, `int` și `long`.

tipuri primitive: formate din tipurile întregi, virgulă mobilă, boolean și caracter.

while: instrucțiune iterativă.

Erori frecvente

1. Adăugarea neintenționată a simbolului `;` imediat după instrucțiunile `for`, `while` conduce la erori logice greu de detectat, deoarece instrucțiunile care se presupune că ar face parte din ciclurile respective, sunt de fapt în afara ciclului, motiv pentru care se vor executa o singură dată. În realitate, `;` reprezintă instrucțiunea vidă, și este tratată de compilator ca orice altă instrucțiune.

2. O metodă care în antet declară că trebuie să returneze o valoare, dar în corpul de definiție nu face acest lucru, conduce la apariția unei erori de compilare.
3. Pentru operații logice ȘI/SAU se folosesc & și |, și nu operatorii & și |.
4. Adăugarea unui 0 în fața unui număr, transformă numărul respectiv în baza 8. De exemplu, dacă numărului 37, îi adăugăm un 0, obținând 037, acesta devine reprezentarea în baza 8 a unui alt număr, 31.
5. Pentru compararea valorilor stocate în variabile, folosiți == și nu =. De exemplu, `if (a == 0)` este corect, în timp ce `if (a = 0)` nu este corect și compilatorul va semnala o eroare (în loc de `boolean` a găsit `int`).
6. Nu stocați într-o variabilă o valoare mai mare/mică decât limitele permise. De exemplu, dacă într-o variabilă de tip `byte`, care poate stoca valori între -128 și 127, se încearcă stocarea valorii 128, atunci vom avea o *depășire* (overflow), situație în care numărul trece de la o extremă la alta, adică -128, și va continua numărătoarea de acolo. Situațiile de depășire a intervalului admis sunt greu de depistat într-un program, de aceea este indicat să vă alegeți un tip de variabilă care să ofere spațiul necesar de stocare (de exemplu, `int` sau `long`).
7. Clauza `else` aparține de cea mai apropiată instrucțiune `if`. Dacă dorim să asociem clauza `else` cu o instrucțiune `if` deschisă anterior, trebuie instrucțiunile din cadrul acelui `if` să fie cuprinse între acolade, ca în exemplul de mai jos:

```

1 if (a>0)
2 {
3   if (b>0)
4     c = a+b ;
5 }
6 else //else va fi asociat cu if(a>0)
7   c = a*b ;

```

8. Numele clasei Java trebuie să fie același cu numele fișierului sursă care o conține; atenție la litere mari și mici!
9. Lipsa instrucțiunii `break` pe ramurile logice ale instrucțiunii `switch` duce la apariția unui efect nedorit în cele mai multe cazuri: de la ramura

curentă se trece la cea următoare care este și ea executată. Ca o consecință se execută mai multe ramuri din cadrul instrucțiunii `switch`.

Exerciții

Pe scurt

1. Ce extensii folosesc fișierele sursă și cele compilate în Java?
2. Descrieți cele trei tipuri de comentarii în Java.
3. Care sunt cele opt tipuri de date primitive în Java?
4. Care este diferența dintre operatorii `*` și `*=` ?
5. Explicați diferența dintre operatorii unari prefixați și cei postfixați.
6. Descrieți cele trei tipuri de instrucțiuni de ciclare în Java.
7. Descrieți toate modurile de utilizare ale instrucțiunii `break`. Ce înseamnă o instrucțiune `break` etichetată?
8. Ce face instrucțiunea `continue`?
9. Ce înseamnă supraîncărcarea de metode?
10. Ce înseamnă transmiterea de parametri prin valoare?

Teorie

1. Fie `b` cu valoarea 5 și `c` cu valoarea 8. Care sunt valorile lui `a`, `b` și `c` după execuția fiecărei linii de cod de mai jos?

```
1 a = b++ + c++ ;
2 a = b++ + ++c ;
3 a = ++b + c++ ;
4 a = ++b + ++c ;
```

2. Care este rezultatul expresiei `true && false || true` ?
3. Dați un exemplu pentru care ciclul `for` de mai jos *nu* este echivalent cu ciclul `while` de mai jos:

```

1 for (init; test; actualizare)
2 {
3     instructiuni;
4 }

1 init;
2 while (test)
3 {
4     instructiuni;
5     actualizare;
6 }

```

Indicație

În cazul în care ciclul `for` conține o instrucțiune continuă, înainte de a trece la următoarea iterație se va executa actualizare.

4. Pentru programul de mai jos, care sunt valorile posibile la afișare?

```

1 public class WhatIsX
2 {
3     public static void f(int x)
4     {
5         /*corpul functiei este necunoscut*/
6     }
7
8     public static void main(String [] args)
9     {
10        int x = 0;
11        f(x);
12        System.out.println(x);
13    }
14 }

```

În practică

1. Scrieți o instrucțiune `while` echivalentă cu ciclul `for` de mai jos. La ce ar putea fi utilizat un astfel de ciclu?

```

for ( ; ; )
    instructiune;

```

2. Scrieți un program care afișează numerele de la 1 la 100. Modificați apoi programul pentru a întrerupe ciclul de afișare după numărul 36 folosind instrucțiunea `break`. Încercați apoi să folosiți `return` în loc de `break`. Care este diferența?

3. Scrieți un program care generează tabelele pentru înmulțirea și adunarea numerelor cu o singură cifră.
4. Scrieți un program care să genereze 25 de valori aleatoare de tip `int`. Pentru fiecare valoare generată folosiți o instrucțiune `if-else` pentru a determina dacă este mai mică, egală sau mai mare decât o altă valoare generată aleator.
5. Scrieți un program care afișează toate numerele prime cu valori cuprinse între 0 și `Integer.MAX_INT`. Folosiți două cicluri `for` (unul pentru fiecare număr și unul pentru testarea divizibilității) și operatorul `%`.
6. Scrieți două metode statice. Prima să returneze maximul a trei numere întregi, iar a doua maximul a patru numere întregi.
7. Scrieți o metodă statică care primește ca parametru un an și returnează `true` dacă anul este bisect și `false` în caz contrar.

3. Referințe

Noi nu reținem zile; reținem doar momente.

Autor anonim

În capitolul 2 am prezentat tipurile primitive din Java. Toate tipurile care nu fac parte dintre cele opt tipuri primitive, inclusiv tipuri importante cum ar fi stringuri, șiruri și fișiere, sunt tipuri referință.

În acest capitol vom învăța:

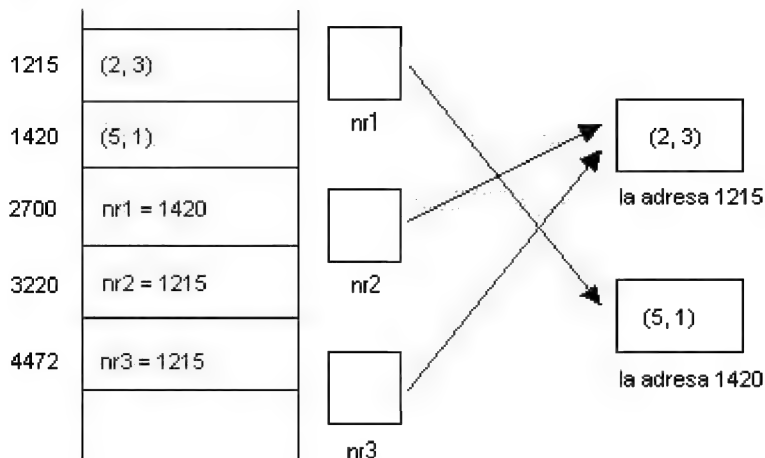
- Ce este un tip referință și ce este o variabilă referință;
- Prin ce diferă un tip referință de un tip primitiv;
- Exemple de tipuri referință, incluzând stringuri și șiruri.

3.1 Ce este o referință?

În capitolul 2 am examinat cele opt tipuri primitive împreună cu câteva operații care pot fi realizate pe variabile având aceste tipuri. Toate celelalte tipuri de date din Java sunt referințe. Ce este deci o referință?

O *variabilă referință* în Java (numită adeseori simplu *referință*) este o variabilă care reține adresa de memorie la care se află un anumit obiect.

Figura 3.1: Ilustrarea unei referințe: Obiectul de tip Complex stocat la adresa de memorie 1215 este referit atât de către nr2 cât și de către nr3. Obiectul de tip Complex stocat la adresa 1420 este referit de către nr1. Locațiile de memorie unde sunt reținute variabilele au fost alese arbitrar.



Ca un exemplu, în **Figura 3.1** există două obiecte de tipul `Complex`¹. Presupunem că aceste obiecte au fost stocate la adresele de memorie 1215 și respectiv 1420. Pentru aceste două obiecte am definit trei referințe, `nr1`, `nr2` și `nr3`. Atât `nr2` cât și `nr3` referă (indică) obiectul stocat la adresa 1215; `nr1` referă obiectul stocat la adresa 1420. Aceasta înseamnă că atât `nr2` cât și `nr3` au valoarea 1215, iar `nr1` va avea valoarea 1420. Rețineți că locațiile efective, cum ar fi 1215 și 1420, sunt atribuite de compilator la discreția sa (unde găsește memorie liberă). În consecință, aceste valori nu sunt utile efectiv ca valori numerice. Totuși, faptul că `nr2` și `nr3` au *aceeași* valoare *este* folositor: înseamnă că ele referă același obiect.

O referință stochează întotdeauna adresa la care un anumit obiect se află, cu excepția situației când nu referă nici un obiect. În acest caz va stoca *referința nulă*, notată în Java cu `null`. Limbajul Java nu permite referințe către tipurile primitive (cum ar fi `int` sau `float`).

Există două categorii distincte de operații care se pot aplica variabilelor referință:

¹ Care conțin partea reală și cea imaginară a unui număr complex.

1. Prima categorie permite examinarea și manipularea valorii referință. De exemplu, dacă modificăm valoarea stocată în `nr1` (care este 1420), putem să facem ca `nr1` să refere un alt obiect. Putem de asemenea compara `nr1` și `nr3` pentru a vedea dacă referă același obiect;
2. A doua categorie de operații se aplică obiectului care este referit. Am putea de exemplu examina sau modifica starea unuia dintre obiectele de tipul `Complex` (am putea examina partea reală și cea imaginară a unui obiect de tipul `Complex`). Accesul la oricare obiect în Java se face exclusiv prin intermediul unei referințe către acel obiect.

Înainte de a descrie ce se poate face cu ajutorul referințelor, să descriem ceea ce nu se poate face. Să considerăm expresia `nr1 * nr2`. Deoarece valorile reținute de `nr1` și `nr2` sunt respectiv 1420 și 1215, produsul lor ar fi 1725300. Totuși acest calcul este complet lipsit de sens și rezultatul său nu are nici o valoare practică. Variabilele referință rețin adrese și nu poate fi asociată nici o semnificație logică înmulțirii adreselor.

Analog, `nr1++` nu are nici un sens în Java; ar sugera ca `nr1` - care are valoarea 1420 - să fie crescut la 1421, dar în acest caz nu ar mai referi un obiect valid. Multe alte limbaje de programare, cum ar fi C, definesc noțiunea de *pointer* care are un comportament similar cu cel al unei variabile referință. Totuși, pointerii în C sunt mult mai periculoși, deoarece este permisă aritmetica pe adresele stocate. În plus, deoarece C permite pointeri și către tipurile primitive trebuie avut grijă pentru a distinge între aritmetica pe adrese și aritmetica pe variabilele care sunt referite. Acest lucru se face prin *dereferențierea* explicită a pointerului. În practică, pointerii limbajului C tind să provoace numeroase erori greu detectabile, care pot adeseori provoca dureri de cap până și programatorilor experimentați!

În Java, singurele operații care sunt permise asupra referințelor (cu o singură excepție pentru Stringuri), sunt atribuirea prin intermediul operatorului `=` și comparația prin intermediul operatorilor `==` și `!=`. De exemplu, prin atribuirea lui `nr3` a valorii lui `nr1`, vom face ca `nr3` să refere același obiect pe care îl referă `nr1`. Acum expresia `nr1 == nr3` este adevărată, deoarece ambele referințe stochează valoarea 1420 și referă deci același obiect. `nr3 != nr2` este de asemenea adevărată, deoarece `nr3` și `nr2` referă acum obiecte distincte.

Cealaltă categorie de operații se referă la obiectul care este referit. Există doar trei acțiuni fundamentale care pot fi realizate:

1. Aplicarea unei conversii de tip;
2. Accesul la un câmp al obiectului sau apelul unei metode prin operatorul punct (`.`);

3. Utilizarea operatorului `instanceof` pentru a verifica dacă obiectul reținut are un anumit tip.

Secțiunea următoare ilustrează mai detaliat operațiile pe referințe.

3.2 Fundamente despre obiecte și referințe

În Java un *obiect* este orice variabilă care nu este de tip primitiv. Obiectele sunt tratate diferit față de tipurile primitive. Variabilele de tipuri primitive sunt manipulate prin valoare, ceea ce înseamnă că valorile lor sunt reținute în acele variabile și sunt copiate dintr-o variabilă primitivă în altă variabilă primitivă în timpul instrucțiunii de atribuire. După cum am arătat în secțiunea anterioară, variabilele referință stochează referințe către obiecte.

Obiectul în sine este stocat undeva în memorie, iar variabila referință stochează adresa de memorie a obiectului. Astfel, variabila referință nu este decât un *nume* pentru acea zonă de memorie. În consecință variabilele primitive și cele referință vor avea un comportament diferit. Prezenta secțiune examinează mai în detaliu aceste diferențe și ilustrează operațiile care sunt permise asupra tipurilor referință.

3.2.1 Operatorul punct (.)

Operatorul punct (.) este folosit pentru a selecta o metodă care se aplică unui obiect. De exemplu, să presupunem că avem un obiect de tip `Cerc` care definește metoda `arie`. Dacă variabila `unCerc` este o referință către un obiect de tip `Cerc`, atunci putem calcula aria cercului referit (și salva această arie într-o variabilă de tip `double`) astfel:

```
double arieCerc = unCerc.arie();
```

Este posibil ca variabila `unCerc` să rețină referința `null`. În acest caz, aplicarea operatorului punct va genera o excepție `NullPointerException` (prezentarea excepțiilor este realizată în capitolul 6) la execuția programului. De obicei această excepție va determina terminarea anormală a programului.

Operatorul punct poate fi folosit și pentru a accesa componentele individuale ale unui obiect, dacă cel care a proiectat obiectul permite acest lucru. Capitolul următor descrie cum se poate face acest lucru și tot acolo vom explica de ce în general este preferabil să nu se permită accesul direct la componentele individuale ale unui obiect.

3.2.2 Declararea obiectelor

Am văzut deja care este sintaxa pentru declararea variabilelor primitive. Pentru obiecte există o diferență importantă. Atunci când declarăm o referință, nu facem decât să furnizăm un nume care poate fi utilizat pentru a referi un obiect stocat în prealabil în memorie. Totuși, declarația în sine *nu* furnizează și acel obiect. Să presupunem, de exemplu, că avem un obiect de tip `Cerc` căruia dorim să îi calculăm aria folosind metoda `arie`. Să considerăm secvența de instrucțiuni de mai jos:

```
1 Cerc unCerc; //unCerc poate referi un obiect de tip Cerc
2 double arieCerc = unCerc.arie(); //calcul arie pentru cerc
```

Totul pare în regulă cu aceste instrucțiuni, până când ne aducem aminte că `unCerc` este numele unui obiect oarecare de tip `Cerc`, dar nu am creat nici un cerc efectiv. În consecință, după ce se declară variabila `unCerc`, aceasta va conține valoarea `null`, ceea ce înseamnă că `unCerc` încă nu referă un obiect `Cerc` valid. Rezultă deci că a doua linie din secvența de cod de mai sus este incorectă, deoarece încercăm să calculăm aria unui cerc care încă nu există (este ca și când am încerca să ne suim la volanul unei mașini pe care încă nu o avem). În exemplul de față chiar compilatorul va detecta eroarea, afirmând că `unCerc` "nu este inițializat". În alte situații mai complexe compilatorul nu va putea detecta eroarea și se va genera o eroare `NullPointerException` doar la execuția programului.

Singura posibilitate (normală) de a alocă memorie unui obiect Java este folosirea cuvântului cheie `new`. `new` este folosit pentru a construi un nou obiect. Astfel, secvența de cod de mai sus corectată este:

```
1 Cerc unCerc; //unCerc poate referi un obiect de tip Cerc
2 unCerc = new Cerc(); //acum unCerc refera un obiect alocat
3 double arieCerc = unCerc.arie(); //calcul arie pentru cerc
```

Remarcați parantezele care se pun după numele obiectului.

Adeseori programatorii combină declararea și inițializarea obiectului ca în exemplul de mai jos:

```
1 Cerc unCerc = new Cerc(); //acum unCerc refera un obiect alocat
2 double arieCerc = unCerc.arie(); //calcul arie pentru cerc
```

Multe obiecte pot fi de asemenea construite cu anumite valori inițiale. De exemplu, obiectul de tip `Cerc` ar putea fi construit cu trei parametri, doi pentru coordonatele centrului și unul pentru lungimea razei.

```
1 //cerc cu centru (0, 0) si de raza 10
2 Cerc unCerc = new Cerc(0, 0, 10);
3 //calcul ariei pentru cerc
4 double arieCerc = unCerc.arie();
```


3.2.3 Colectarea de gunoaie (garbage collection)

Deoarece toate obiectele trebuie construite, ne-am putea aștepta ca atunci când nu mai este nevoie de ele să trebuiască să le distrugem. Totuși, în Java, când un obiect din memorie nu mai este referit de nici o variabilă, memoria pe care o consumă va fi eliberată automat. Această tehnică se numește *colectare de gunoaie*.

3.2.4 Semnificația operatorului =

Să presupunem că avem două variabile de tipuri primitive, *x* și *y*. În această situație, semnificația instrucțiunii de atribuire

```
x = y;
```

este simplă: valoarea stocată în *y* este transferată în variabila primitivă *x*. Modificările ulterioare ale lui *y* nu afectează în nici un fel valoarea lui *x*.

Pentru referințe, semnificația lui = este *exact* aceeași: se copiază valorile referințelor, adică adresele pe care acestea le indică. Dacă *x* și *y* sunt referințe (de tipuri compatibile), atunci, după operația de atribuire, *x* va referi același obiect ca și *y*. Ceea ce se copiază în acest caz sunt adrese. Obiectul pe care *x* îl referea înainte nu mai este referit de *x*. Dacă *x* a fost singura referință către acel obiect, atunci obiectul nu mai este referit acum de nici o variabilă și este disponibil pentru colectarea de gunoaie. Rețineți faptul că obiectele nu se copiază prin operatorul =.

Iată câteva exemple. Să presupunem că dorim să creăm două obiecte de tip *Cerc* pentru a calcula suma ariilor lor. Creăm mai întâi obiectul *cerc1*, după care încercăm să creăm obiectul *cerc2* prin modificarea lui *cerc1* după cum urmează (vezi și **Figura 3.2**):

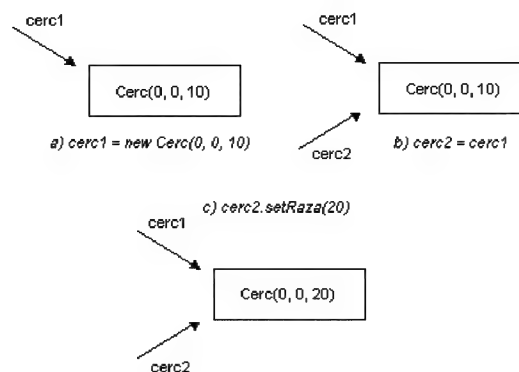
```
1 Cerc cerc1 = new Cerc(0, 0, 10); //un cerc de raza 10
2 Cerc cerc2 = cerc1;
3 cerc2.setRaza(20); // modificam raza la 20
4 double arieCercuri = cerc1.arie() + cerc2.arie(); //calcul arie
```

Acest cod nu va funcționa corect, deoarece nu s-a construit decât un singur obiect de tip *Cerc*. Astfel, cea de-a doua instrucțiune nu face decât să spună că *cerc2* este un alt nume pentru *cerc1*, construit anterior. Cercul construit în prima linie are acum două nume. A treia instrucțiune modifică raza cercului la 20, dar de fapt se modifică raza unicului cerc creat, deci ultima linie adună aria aceluiasi cerc de rază 20.

Secvența de cod corectă este:

```
1 Cerc cerc1 = new Cerc(0, 0, 10); //un cerc de raza 10
```

Figura 3.2: `cerc1` și `cerc2` indică același obiect. Modificarea razei lui `cerc2` implică și modificarea razei lui `cerc1`.



```
2 Cerc cerc2 = new Cerc();
3 cerc2.setRaza(20); // modificam raza la 20
4 double arieCercuri = cerc1.arie() + cerc2.arie(); // calcul arie
```

La o primă vedere, faptul că obiectele nu pot fi copiate, pare să fie o limitare severă. În realitate însă nu este deloc așa, deși ne trebuie un pic de timp pentru a ne obișnui cu acest lucru. Există totuși anumite situații când trebuie să copiem obiecte; în aceste situații se va folosi metoda `clone()`. `clone()` folosește `new()` pentru a crea un nou obiect duplicat. În această carte metoda `clone()` nu este folosită.

3.2.5 Transmiterea de parametri

Din cauza faptului că apelul se face prin valoare, parametrii actuali (de apel) se transpun în parametri formali folosind atribuirea obișnuită. Dacă parametrul trimis este un tip referință, atunci știm deja că prin atribuire atât parametrul formal, cât și parametrul de apel vor referi același obiect. Orice metodă aplicată parametrului formal este astfel implicit aplicată și parametrului de apel. În alte limbaje de programare acest tip de apel se numește *apelare prin referință*. Utilizarea acestei noțiuni în Java ar fi oarecum nepotrivită, deoarece ne-ar putea face să credem că transmiterea referințelor s-ar face în mod diferit de tipurile primitive. În realitate, transmiterea parametrilor nu s-a modificat; ceea ce s-a modificat sunt *parametrii* în sine, care nu mai sunt tipuri primitive, ci tipuri referință. Metoda `f` de mai jos primește ca parametri un obiect de tip `Cerc` și o valoare întreagă.

```

1 public void f(Cerc unCerc, int val)
2 {
3     unCerc.setRaza(val); //modifica raza obiectului apelant
4     val += 5; //nu are nici un efect asupra parametrului actual
5 }

```

Secvența de cod

```

1 int v = 5;
2 Cerc cerc = new Cerc();
3 f(cerc, v);

```

va avea ca efect modificarea razei obiectului `cerc` la 5, dar valoarea lui `v` va rămâne nemodificată. Explicație acestui fenomen este simplă: `cerc` și `unCerc` sunt variabile referință *distincte* care indică *același* obiect, în timp ce `v` și `val` sunt variabile de tip primitiv *distincte*, și în consecință modificarea uneia nu are nici un efect asupra celeilalte. Pentru a fi mai clar, ne putem imagina că la apelul metodei `f()`, au loc atribuiri

```

1 unCerc = cerc;
2 v = val;

```

după care se execută secvența de instrucțiuni a metodei.

3.2.6 Semnificația operatorului ==

Pentru tipurile primitive operația `==` are valoarea `true` dacă acestea au valori identice. Pentru tipuri referință semnificația lui `==` este diferită, dar perfect consistentă cu discuția din paragraful anterior.

Două variabile referință sunt egale via `==` dacă ele referă același obiect (s-au ambele sunt `null`). Să considerăm următorul exemplu:

```

1 Cerc cerc1 = new Cerc(0, 0, 10); //un cerc de raza 10
2 Cerc cerc2 = new Cerc(0, 0, 10); //un alt cerc tot de raza 10
3 Cerc cerc3 = cerc2;

```

În acest caz avem două obiecte. Primul este cunoscut sub numele de `cerc1`, al doilea este cunoscut sub două nume: `cerc2` și `cerc3`. Expresia `cerc2 == cerc3` este adevărată. Totuși, deși `cerc1` și `cerc2` referă obiecte care au valori egale, expresia `cerc1 == cerc2` este falsă. Chiar dacă cele două obiecte au valori egale (ambele sunt cercuri cu centrul în origine și rază de 10), ele sunt entități distincte. Aceleași reguli se aplică și pentru operatorul `!=`.

Cum facem însă pentru a vedea dacă obiectele referite sunt identice? De exemplu, cum putem să verificăm faptul că `cerc1` și `cerc2` referă obiecte `Cerc` care sunt identice? Obiectele pot fi comparate folosind metoda `equals`. Vom vedea în curând un exemplu de folosire a lui `equals`, în momentul în

care vom discuta despre tipul `String`. Fiecare obiect are o metodă `equals`, care, în mod implicit, nu face altceva decât testul `==` descris mai sus. Pentru ca `equals` să funcționeze corect, programatorul trebuie să redefinească această metodă pentru obiectele pe care le crează (redefinirea metodelor va fi prezentată în capitolul următor).

3.2.7 Supraîncărcarea operatorilor pentru obiecte

În afara unei singure excepții pe care o vom discuta în paragraful următor (concatenarea de șiruri), operatorii nu pot fi definiți pentru a lucra cu obiecte². Astfel, nu există operatorul `<` pentru nici un fel de obiect. Pentru acest scop, va trebui definită o metodă, cum ar fi `lessThan`, care va realiza comparația.

3.3 Șiruri de caractere (stringuri)

Șirurile de caractere în Java sunt definite folosind tipul `String`. Limbajul Java face să pară că `String` este un tip primitiv, deoarece pentru el sunt definiți operatorii `+` și `+=` pentru concatenare (am arătat în paragraful anterior că operatorii nu pot fi aplicați obiectelor). Totuși, acesta este singurul tip referință pentru care Java a permis supraîncărcarea operatorilor (pentru comoditatea scrierii programelor). În rest, `String` se comportă ca orice alt obiect.

3.3.1 Fundamentele utilizării stringurilor

Există două reguli fundamentale referitoare la obiectele de tip `String`. Prima este aceea că, exceptând operatorul de concatenare, obiectele `String` se comportă ca toate celelalte obiecte. A doua regulă este aceea că stringurile sunt ne-modificabile. Aceasta înseamnă că, odată construit, un obiect de tip `String` nu mai poate fi modificat.

Deoarece obiectele de tip `String` nu se pot modifica, putem folosi liniștiți operatorul `=` pentru ele, fără nici un risc. Iată un exemplu:

```
1 String vid = "";
2 String mesaj = "Salutare!";
3 String repetat = mesaj;
```

După aceste declarații, există două obiecte de tip `String`. Primul este un șir `vid` și este referit de variabila `vid`, iar al doilea este șirul `"Salutare!"`,

²Aceasta este o diferență notabilă între Java și C++, care permite supraîncărcarea operatorilor pentru obiecte. Inginerii de la Sun au considerat că supraîncărcarea operatorilor pentru obiecte aduce mai multe probleme decât beneficii și au decis ca Java să nu permită acest lucru.

care este referit de variabilele mesaj și repetat. Pentru majoritatea obiectelor, faptul că obiectul este referit de două variabile ar putea genera probleme. Totuși, deoarece stringurile nu pot fi modificate, partajarea lor nu pune nici un fel de probleme. Singura posibilitate de a modifica valoarea către care referă variabila repetat este aceea de a construi un nou obiect de tip `String` și a-l atribui lui repetat. Această operație nu va avea nici un efect asupra valorii pe care o referă mesaj.

3.3.2 Concatenarea stringurilor

Java nu permite supraîncărcarea operatorilor pentru tipurile referință. Totuși, pentru comoditate, se acordă o excepție specială pentru concatenarea obiectelor de tipul `String`.

Atunci când cel puțin unul dintre operanzi este de tip `String`, operatorul `+` realizează concatenarea. Rezultatul este o referință către un obiect nou construit de tip `String`. Iată câteva exemple:

```
1 "Sunt" + " curajos!" //-> "Sunt curajos!"
2 2 + " mere"         //-> "2 mere ", 2 este convertit la String
3 "mere " + 2         //-> "mere 2"
4 "a" + "b" + "c"     //-> "abc"
```

Șirurile de caractere formate dintr-un singur caracter NU trebuie înlocuite cu constante de tip caracter (constantele caracter sunt de fapt numere).

Java dispune și de operatorul `+=` pentru șiruri de caractere. Efectul instrucțiunii `str += expr` este `str = str + expr`. Cu alte cuvinte `str` va referi un nou `String` generat de `str + expr`.

Iată un exemplu:

```
1 String s1 = "ab";
2 String s2 = "cd";
3 s1 += s2; //s1 va deveni "abcd"
4 System.out.println(s1); //va afisa "abcd"
```

Este important să observăm că între atribuirea:

```
i = i + 5 //i este un intreg
```

și atribuirea:

```
str = str + "hello" //str este un String
```

există o diferență esențială. În primul caz, variabila `i` este incrementată cu 5; locația de memorie a lui `i` nu se modifică. În al doilea caz, se crează un nou string având valoarea `str + "hello"`. După atribuire, `str` va referi acest nou string. Fostul string referit va fi supus colectării de gunoaie (*garbage-collection*) dacă nu a existat o altă referință către el.

3.3.3 Compararea stringurilor

Deoarece operatorul de adunare funcționează pe șiruri de caractere, am fi tentați să credem că funcționează și operatorii relaționali. Acest lucru nu este însă adevărat.

Conform regulii privind supraîncărcarea operatorilor, operatorii relaționali ($<$, $<=$, $>$, $>=$) nu sunt definiți pentru obiecte de tip `String`. Mai mult, operatorii `==` și `!=` au semnificația clasică pentru obiecte de tip referință (compară adrese și nu obiecte). De exemplu, pentru două obiecte de tip `String`, x și y , expresia $x == y$ este adevărată doar dacă x și y referă același obiect de tip `String`. Astfel, dacă x și y referă obiecte diferite cu conținut identic, expresia $x == y$ este falsă. Același raționament este valabil și pentru `!=`.

Pentru a testa egalitatea a două obiecte de tip `String`, se folosește metoda `equals`. Expresia `x.equals(y)` este adevărată dacă șirurile de caractere referite de x și de y sunt identice.

Ca exemplu să considerăm următoarele șiruri:

```
1 String x = "ab";
2 String y = "ab";
3 String z = y;
```

În această situație, expresia $x==y$ va fi falsă pentru că x și y sunt referințe către două obiecte diferite (chiar dacă se întâmplă ca acestea să aibe același conținut, și anume "ab"), în timp ce expresia $y == z$ va fi adevărată pentru că y și z sunt referințe către același obiect. Pentru a compara cele două șiruri de caractere referite de x și y , se folosește metoda `equals()`. În concluzie, `x.equals(y)` va fi o expresie adevărată.

Un test mai general poate fi realizat cu metoda `compareTo()`. Utilizând expresia `x.compareTo(y)`, se compară două obiecte de tip `String`, x și y . Valoarea returnată este un număr negativ, zero sau un număr pozitiv dacă x este mai mic, egal, respectiv mai mare decât y din punct de vedere al ordinii lexicografice.

Compararea stringurilor poate fi realizată și cu ajutorul unei alte metode, numită `compareToIgnoreCase()`. Această metodă are același comportament cu `compareTo()`, doar că metoda `compareToIgnoreCase()` nu face distincție între literele mici și literele mari ale alfabetului.

De exemplu, fie:

```
1 String s1 = "AbcD";
2 String s2 = "abcd";
```

În acest caz, expresia `s1.compareTo(s2)` va fi falsă, pentru că "A" este diferit de "a", în timp ce expresia `s1.compareToIgnoreCase(s2)` va fi adevărată pentru că metoda `compareToIgnoreCase` nu face diferența între

literele mari și cele mici. Altfel spus, din punctul de vedere al metodei `compareToIgnoreCase`, "A" este egal cu "a". Este evident că metoda `compareToIgnoreCase` nu are sens să fie folosită pe șiruri de caractere care nu conțin litere, ca în exemplul următor `s1.compareToIgnoreCase("123")`.

3.3.4 Alte metode pentru stringuri

Lungimea unui obiect de tip `String` (un șir vid are lungimea 0) poate fi obținută cu metoda `length()`. Deoarece `length()` este o metodă, în momentul apelului parantezele sunt necesare.

Există două metode pentru a accesa caracterele din interiorul unui `String`. Metoda `charAt` returnează caracterul aflat la poziția specificată (primul caracter este pe poziția 0). Metoda `substring` returnează o referință către un `String` nou construit. Metoda are ca parametri poziția de început și poziția primului caracter neinclus.

Iată un exemplu de folosire a acestor metode:

```
1 String mesaj = "Hello";
2 int lungimeMesaj = mesaj.length();           //lungimea este 5
3 char ch = mesaj.charAt(1);                   //ch este 'e'
4 String subSir = mesaj.substring(2, 4);       //sub este "ll"
```

Limbajul Java oferă o paletă impresionantă de metode pentru manipularea șirurilor de caractere. Pe lângă cele prezentate anterior, metode des utilizate sunt de asemenea:

- `concat()`

Descriere: metoda `concat` adaugă la sfârșitul unui șir de caractere un alt șir de caractere. Deoarece un obiect de tip `String` nu este modificabil (mutabil), prin concatenare se va crea un nou obiect, ce va fi referit de variabila referință `s1`. Fostul obiect referit de `s1` va fi supus colectării de gunoaie, în situația în care nici o altă variabilă nu îl referă.

Antet: `String concat(String str)`

Exemplu:

```
1 String s1 = "a";
2 s1.concat("b"); //s1 devine "ab"
```

- `endsWith()`

Descriere: verifică dacă șirul de caractere are ca sufix un alt șir de caractere.

Antet: `boolean endsWith(String sufix)`

Exemplu:

```
1 String s1 = "abcde";
2
3 System.out.println(s1.endsWith("de"));
4 /*
5  * va afisa true, pentru ca sirul de caractere "abcde" se
6  * termina cu sirul de caractere "de"
7  */
```

- `equalsIgnoreCase()`

Descriere: verifică dacă două șiruri sunt egale, fără a face diferență între literele mari și cele mici ale alfabetului.

Antet: `boolean equalsIgnoreCase(String altString)`

Exemplu:

```
1 String s1 = "ABc";
2
3 System.out.println(s1.equalsIgnoreCase("abc"));
4 /*
5  * va afisa true, pentru ca nu se face diferenta intre
6  * literele mici si cele mari ale alfabetului
7  */
```

- `getBytes()`

Descriere: convertește stringul într-un șir de bytes(octeți), astfel încât fiecărui caracter din șir îi va corespunde un întreg de tip `byte`, reprezentând primul byte din codul Unicode al caracterului respectiv³.

Antet: `byte[] getBytes()`

Exemplu:

```
1 byte[] b1 = "abcd".getBytes();
2 /*
3  * sirul b1 va contine codurile Unicode ale caracterelor
4  * ce formeaza sirul "abcd", adica 97, 98, 99, 100
5  */
```

³În realitate, procesul este ceva mai complex: metoda `getBytes()` va transforma codul Unicode al fiecărui caracter din șir (având 2 octeți) în codul pe 1 octet specific platformei pe care rulează mașina virtuală.

- `indexOf()`

Descriere: returnează poziția la care se află prima apariție a unui subșir într-un șir. Dacă subșirul nu se regăsește în șirul respectiv, metoda returnează valoarea -1.

Antet: `int indexOf(String str)`

Exemplu:

```
1 String s1 = "Limbajul Java este orientat pe obiecte";
2 String s2 = "Java";
3 String s3 = "C++";
4
5 System.out.println(s1.indexOf(s2));
6 // va afisa 9 (indicele din s1 la care incepe subsirul s2)
7
8 System.out.println(s1.indexOf(s3));
9 // va afisa -1 (sirul s1 nu contine subsirul s3)
```

- `lastIndexOf()`

Descriere: returnează poziția la care se află ultima apariție a unui subșir într-un șir. Dacă subșirul nu se află în șirul respectiv, metoda returnează valoarea -1.

Antet: `int lastIndexOf(String str)`

Exemplu:

```
1 String s1 = "El invata Java pentru ca Java e OOP";
2 String s2 = "Java";
3
4 System.out.println(s1.lastIndexOf(s2));
5 /*
6  * va afisa 25 (indicele la care incepe ultima aparitie
7  * a subsirului s2 in sirul s1)
8  */
```

- `replace()`

Descriere: înlocuiește aparițiile unui caracter într-un șir cu un alt caracter și returnează noul șir astfel format. Dacă respectivul caracter nu apare în șir, atunci noul șir va fi la fel cu cel inițial.

Antet: `String replace(char carVechi, char carNou)`

Exemplu:

```
1 String s1 = "Java";
2 String s2 = s1.replace('a', 'i');
3 // s2 va avea valoarea "Jivi"
```

- `startsWith()`

Descriere: verifică dacă un șir de caractere are ca prefix un alt șir de caractere.

Antet: `boolean startsWith(String prefix)`

Exemplu:

```
1 String s1 = "abcd";
2 System.out.println(s1.startsWith("abc"));
3 /*
4  * va afisa true pentru ca stringul s1 incepe
5  * cu sirul de caractere "abc"
6  */
```

- `toLowerCase()`

Descriere: convertește toate literele mari din string în litere mici.

Antet: `String toLowerCase()`

Exemplu:

```
1 String s1 = "ABCD E";
2 String s2 = s1.toLowerCase();
3 //s2 va fi egal cu "abcde"
```

- `toUpperCase()`

Descriere: este opusa metodei `toLowerCase`. Convertește toate literele mici în litere mari.

Antet: `String toUpperCase()`

Exemplu:

```
1 String s1 = "abCdE";
2 String s2 = s1.toUpperCase();
3 //s2 va fi egal cu "ABDCE"
```

- `trim()`

Descriere: elimină spațiile de la începutul și sfârșitul unui șir de caractere.

Antet: `String trim()`

Exemplu:

```
1 String s1 = "   Java   ";
2 s1 = s1.trim();
3 //s1 va fi egal cu "Java"
```

3.3.5 Conversia de la string la tipurile primitive și invers

Metoda `toString()` poate fi utilizată pentru a converti orice tip primitiv la `String`. De exemplu, `toString(45)` returnează o referință către șirul nou construit "45". Majoritatea obiectelor furnizează o implementare a metodei `toString()`. De fapt, atunci când operatorul `+` are un operand de tip `String`, operandul care nu este de tip `String` este automat convertit la `String` folosind metoda `toString()`. Pentru tipurile de date numerice, există o variantă a metodei `toString()` care permite precizarea unei anumite baze. Astfel, instrucțiunea:

```
System.out.println(Integer.toString(55, 2));
```

are ca efect tipărirea reprezentării în baza 2 a numărului 55.

Pentru a converti un `String` la un `int` există posibilitatea de a folosi metoda `Integer.parseInt()`. Această metodă generează o excepție dacă `String`-ul convertit nu conține o valoare întreagă. Pentru a obține un `double` dintr-un `String` se poate utiliza metoda `parseDouble()`. Iată două exemple:

```
1 int x = Integer.parseInt("75");
2 double y = Double.parseDouble("3.14");
```

3.4 Șiruri

Șirurile⁴ sunt structura fundamentală prin care se pot reține mai multe elemente de același tip. În Java, șirurile nu sunt tipuri primitive; ele se comportă foarte asemănător cu un obiect. Din acest motiv, multe dintre regulile care sunt valabile pentru obiecte se aplică și la șiruri.

Fiecare element dintr-un șir poate fi accesat prin mecanismul de *indiciere* oferit de operatorul `[]`. Spre deosebire de limbajele C sau C++, Java verifică validitatea indicilor⁵.

În Java, ca și în C, șirurile sunt întotdeauna indiciate de la 0. Astfel, un șir a cu 3 elemente este format din `a[0]`, `a[1]`, `a[2]`. Numărul de elemente care pot fi stocate în șirul `a` este permanent reținut în variabila `a.length`. Observați că aici (spre deosebire de `String`-uri) nu se pun paranteze. O parcurgere tipică a unui șir este:

⁴Desemnate în alte lucrări și sub numele de *tablouri*, *vectori*, *blocuri*.

⁵Acesta este un lucru foarte important care vine în ajutorul programatorilor, mai ales a celor începători. Indicii a căror valoare depășește numărul de elemente alocat, sunt adeseori cauza multor erori obscure în C și C++. În Java, accesarea unui șir cu un indice în afara limitei este imediat semnalată prin excepția `ArrayIndexOutOfBoundsException`.

```
for (int i = 0; i < a.length; i++)
```

3.4.1 Declarație, atribuire și metode

Un șir de elemente întregi se declară astfel:

```
int [] sir1;
```

Deoarece un șir este un obiect, declarația de mai sus nu alocă memorie pentru șir. Variabila `sir1` este doar un nume (referință) pentru un șir de numere întregi, și în acest moment valoarea ei este `null`. Pentru a alocă 100 de numere întregi, vom folosi instrucțiunea:

```
sir1 = new int[100];
```

Acum `sir1` este o referință către un șir de 100 de numere întregi.

Există și alte posibilități de a declara șiruri. De exemplu, putem unifica declarația și alocarea de memorie într-o singură instrucțiune:

```
int [] sir1 = new int[100];
```

Se pot folosi și liste de inițializare, ca în C sau C++. În exemplul următor se alocă un șir cu patru elemente, care va fi referit de către variabila `sir2`:

```
int [] sir2 = {3, 4, 6, 19};
```

Parantezele pătrate de la declarare pot fi puse fie înainte, fie după numele șirului. Plasarea parantezelor înainte de nume face mai vizibil faptul că este vorba de un șir, de aceea vom folosi această notăție.

Declararea unui șir de obiecte (deci nu tipuri primitive) folosește aceeași sintaxă. Trebuie să rețineți însă că după alocarea șirului, fiecare element din șir va avea valoarea `null`. Pentru fiecare element din șir trebuie alocată memorie separat. De exemplu, un șir cu 5 cercuri se construiește astfel:

```
1 //declaram un sir de cercuri
2 Cerc [] sirDeCercuri;
3
4 //alocam memorie pentru 5 referinte la Cerc
5 sirDeCercuri = new Cerc[5];
6
7 for (int i = 0; i < sirDeCercuri.length; ++i)
8 {
9     sirDeCercuri[i] = new Cerc();
10    //se alocă un obiect Cerc referintei nr. i
11 }
```

Programul din **Listing 3.1** ilustrează modul de folosire al șirurilor în Java. În jocul de loterie se selectează săptămânal șase numere de la 1 la 49. Programul

alege aleator numere pentru 1000 de jocuri și afișează apoi de câte ori a apărut fiecare număr în cele 1000 de jocuri. Linia 14 declară un șir de numere întregi care reține de câte ori a fost extras fiecare număr. Deoarece indiciera șirurilor începe de la 0, adunarea cu 1 este esențială. Fără această adunare am fi avut un șir cu elemente de la 0 la 48, și orice acces la elementul cu indicele 49 ar fi generat o excepție `ArrayIndexOutOfBoundsException` (este adevărat că procedând în acest mod, elementul de pe poziția 0 a rămas neutilizat). Ciclul din liniile 15-18 inițializează valorile șirului cu 0. Restul programului este relativ simplu. Se folosește din nou metoda `Math.random()` care generează un număr în intervalul $[0, 1)$. Rezultatele sunt afișate în liniile 28-31.

Listing 3.1: Program demonstrativ pentru șiruri

```

1 //clasa demonstrativa pentru siruri
2 public class Loterie
3 {
4     //genereaza numere de loterie intre 1 si 49
5     //afiseaza numarul de aparitii al fiecarui numar
6     //declaratii constante:
7     public static final int NUMERE = 49;
8     public static final int NUMERE_PE_JOC = 6;
9     public static final int JOCURI = 1000;
10    //main
11    public static final void main(String [] args)
12    {
13        //genereaza numerele
14        int [] numere = new int[NUMERE + 1];
15        for (int i = 0; i < numere.length; ++i)
16        {
17            numere[i] = 0;
18        }
19
20        for (int i = 0; i < JOCURI; ++i)
21        {
22            for (int j = 0 ; j < NUMERE_PE_JOC; ++j)
23            {
24                numere[(int) (Math.random() * 49) + 1]++;
25            }
26        }
27        //afisare rezultate
28        for (int k = 1; k <= NUMERE; ++k)
29        {
30            System.out.println(k + ": " + numere[k]);
31        }
32    }
33 }

```

Dat fiind faptul că șirul este un tip referință, operatorul = nu copiază șiruri. De aceea dacă *x* și *y* sunt șiruri, efectul secvenței de instrucțiuni:

```
1 int[] x = new int[100];
2 int[] y = new int[100];
3 ...
4 x = y;
```

este că *x* și *y* referă acum al doilea șir.

Șirurile pot fi utilizate ca parametri pentru metode. Regulile de transmitere se deduc logic din faptul că șirul este o referință. Să presupunem că avem o metodă *f* care acceptă un șir de *int* ca parametru. Apelul și definirea arată astfel:

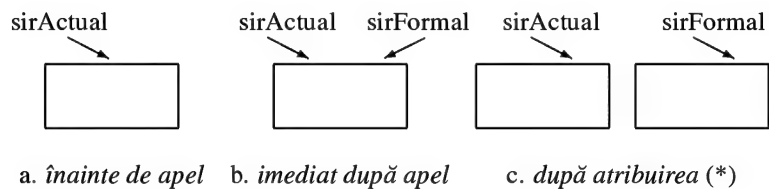
```
1 f(sirActual); //apelul metodei
2 void f(int[] sirFormal); //declaratia metodei
```

Conform convențiilor de transmitere a parametrilor în Java pentru tipurile referință, variabilele *sirActual* și *sirFormal* referă același obiect (la transmiterea parametrului se copiază *valoarea* lui *sirActual*, care este o simplă adresă). Astfel, accesul la elementul *sirFormal*[*i*] este de fapt un acces la elementul *sirActual*[*i*]. Aceasta înseamnă că variabilele conținute în șir pot fi modificate de către metodă. O observație importantă este aceea că linia de cod din cadrul metodei *f*:

```
sirFormal = new int [20]; // (*)
```

nu are nici un efect asupra lui *sirActual*. Acest lucru se datorează faptului că în Java transmiterea parametrilor se face prin valoare, iar la revenirea din funcție adresa pe care o referă *sirActual* rămâne nemodificată. Instrucțiunea de mai sus nu face decât să schimbe șirul către care referă *sirFormal* (vezi figura următoare).

Figura 3.3: Transmiterea parametrilor în Java



Deoarece numele șirurilor sunt doar niște referințe, o funcție poate să returneze un șir.

3.4.2 Expansiunea dinamică a șirurilor

Să presupunem că dorim să citim o secvență de stringuri și să o reținem într-un șir. Una dintre proprietățile fundamentale ale șirurilor este aceea că înainte de a fi utilizate, trebuie să alocăm memorie pentru un număr fix de elemente care vor fi stocate. Dacă nu știm de la bun început câte elemente vor fi stocate în șir, va fi dificil să alegem o valoare rezonabilă pentru dimensiunea șirului. Această secțiune prezintă o metodă prin care putem extinde dinamic șirul dacă dimensiunea inițială se dovedește a fi prea mică. Această tehnică poartă numele de *expansiune dinamică a șirurilor* și permite alocarea de șiruri de dimensiune arbitrară pe care le putem redimensiona pe măsură ce programul rulează.

Alocarea obișnuită de memorie pentru șiruri se realizează astfel:

```
String [] a = new String [10];
```

Să presupunem că după ce am făcut această declarație, la un moment dat avem nevoie de 12 elemente și nu de 10. În această situație putem folosi următoarea manevră:

```
1 String [] original = a;           //salvam referinta lui a
2 a = new String [12];             //alocam din nou memorie
3 for (int i = 0; i < 10; i++)      //copiem elementele in a
4 {
5     a[i] = original[i];
6 }
```

Un moment de gândire este suficient pentru a ne convinge că această operație este consumatoare de resurse, deoarece șirul original trebuie copiat înapoi în noul șir *a*. De exemplu, dacă extensia dinamică a numărului de elemente ar trebui făcută ca răspuns la citirea de date, ar fi inefficient să expansionăm ori de câte ori citim câteva elemente. Din acest motiv, de câte ori se realizează o extensie dinamică, numărul de elemente este crescut cu un coeficient *multiplativ*. Am putea de exemplu dubla numărul de elemente la fiecare expansiune dinamică. Astfel, dintr-un șir cu *N* elemente, generăm un șir cu *2N* elemente, iar costul expansiunii este împărțit între cele *N* elemente care pot fi inserate în șir fără a realiza extensia.

Pentru ca lucrurile să fie concrete, **Listing 3.2** prezintă un program care citește un număr nelimitat de stringuri de la tastatură și le reține într-un șir a cărui dimensiune este extinsă dinamic. Oprirea din citire se realizează în momentul în care ultimul string introdus este egal cu stringul "end". Funcția

`resize()` realizează expansiunea (sau contracția!) șirului returnând o referință către un șir nou construit. Similar, metoda `getStrings()` returnează o referință către șirul în care sunt citite elementele.

La începutul lui `getStrings()`, `nrElemente` este inițializat cu 0 și alocăm memorie pentru 5 elemente. În linia 34 citim în mod repetat câte un element. Dacă șirul este "umplut", lucru indicat de intrarea în testul de la linia 36, atunci șirul este expansionat prin apelul metodei `resize()`. Liniile 64-74 realizează expansiunea șirului folosind strategia prezentată anterior. La linia 42, elementul citit este stocat în tablou, iar numărul de elemente citite este incrementat. În final, în linia 51 contractăm șirul la numărul de elemente citite efectiv. Linia 1 conține o noțiune nouă, directiva `import`, care va fi prezentată pe larg în secțiunea 4.4. De asemenea liniile 31 și 45 conțin instrucțiunea `try-catch`, care va fi prezentată în cadrul capitolului 6.

Listing 3.2: Program pentru citirea unui număr nelimitat de stringuri urmată de afișarea lor

```

1 import java.io.* ;
2 /** Citirea unui numar nelimitat de stringuri. */
3 public class ReadStrings
4 {
5     public static void main(String[] args)
6     {
7         String[] array = getStrings();
8         printSingleDimensionalArray(array);
9
10        char[][] characters = stringsToChars(array);
11        printMultiDimensionalArray(characters);
12    }
13
14    /**
15     * Citeste un numar nelimitat de stringuri
16     * pana intalneste stringul "end",
17     * fara a trata erorile
18     */
19    public static String[] getStrings()
20    {
21        //BufferedReader este prezentata in cap. Java I/O
22        BufferedReader in = new BufferedReader(
23            new InputStreamReader(System.in));
24
25        String[] elemente = new String[5]; //se alocă 5 elemente
26        int nrElemente = 0; //numarul de elemente citite
27        String s; //sir in care se citeste cate o linie
28
29        System.out.println("Introduceti stringuri:");
30    }

```



```

31     try
32     {
33         //cat timp linia este diferita de "end"
34         while (!"end".equalsIgnoreCase(s = in.readLine()))
35         {
36             if (nrElemente == elemente.length)
37             {
38                 //dubleaza dimensiunea sirului "umplut"
39                 elemente = resize(elemente,
40                 elemente.length * 2);
41             }
42             elemente[nrElemente++] = s;
43         }
44     }
45     catch (Exception e)
46     {
47         //nu se trateaza exceptia
48     }
49     System.out.println("Citire incheiata.");
50
51     return resize(elemente, nrElemente);
52     //trunchiaza sirul la numarul de elemente citite
53 }
54
55 /** Afiseaza sirul de stringuri citit. */
56 public static void printSingleDimensionalArray (String[] array)
57 {
58     System.out.println("Elementele citite sunt:");
59     for (int i = 0; i < array.length; i++)
60         System.out.println(array[i]);
61 }
62
63 /** Redimensioneaza sirul. */
64 public static String[] resize(String[] sir, int dimensiuneNoua)
65 {
66     int elementeDeCopiat = Math.min(sir.length,
67     dimensiuneNoua);
68     String[] sirNou = new String[dimensiuneNoua];
69     for (int i = 0; i < elementeDeCopiat; ++i)
70     {
71         sirNou[i] = sir[i];
72     }
73     return sirNou;
74 }
75
76 /** Transforma String[] in char[][]. */
77 public static char[][] stringsToChars (String[] array)
78 {
79     char[][] characters = new char[array.length][5];
80     for (int i = 0; i < array.length; i++)

```

```

81     {
82         characters[i] = new char[array[i].length()];
83         array[i].getChars(0, characters[i].length, characters[i], 0);
84     }
85
86     return characters;
87 }
88
89 /** Afiseaza o matrice de caractere. */
90 public static void printMultiDimensionalArray (
91     char[][] characters)
92 {
93     System.out.println("Elemente sub forma de caractere:");
94
95     for (int i = 0; i < characters.length; i++)
96     {
97         for (int j = 0; j < characters[i].length; j++)
98             System.out.print(characters[i][j]);
99         System.out.println();
100     }
101 }
102 }

```

3.4.3 Șiruri cu mai multe dimensiuni

În anumite situații trebuie să stocăm datele în șiruri cu mai multe dimensiuni. Cel mai des folosite sunt matricele, adică șirurile cu două dimensiuni. Alocarea de memorie pentru șiruri cu mai multe dimensiuni se realizează precizând numărul de elemente pentru fiecare indice, iar indiciera se face plasând fiecare indice între paranteze pătrate. Ca un exemplu, declarația:

```
int[][] x = new int[2][3];
```

definește matricea x, în care primul indice poate fi 0 sau 1, iar al doilea poate lua valori de la 0 la 2.

Listing 3.2 oferă un exemplu simplu de utilizare a unei matrici. După cum se poate observa la linia 10, șirul de stringuri ce a fost citit de la tastatură este transformat într-o matrice de caractere (câte un șir de caractere pentru fiecare string în parte). Metoda `printMultiDimensionalArray` prezintă modul de parcurgere a elementelor matricii obținute.

3.4.4 Argumente în linie de comandă

Parametrii transmiși în linie de comandă sunt disponibili în cadrul unei aplicații, prin examinarea parametrilor funcției `main()`. Șirul de stringuri numit

`args` din funcția `main` conține parametrii transmiși programului Java în linia de comandă. De exemplu, dacă avem un program numit `Suma.java` pe care îl executăm cu comanda:

```
java Suma 2 3
```

atunci parametrul `args[0]` va fi o referință către stringul `"2"`, iar parametrul `args[1]` va fi o referință către `"3"`. Astfel, programul următor implementează o comandă de adunare a numerelor trimise ca argument:

Listing 3.3: Utilizarea argumentelor în linie de comandă

```
1 public class Suma
2 {
3     //afiseaza suma parametrilor primiti in linie de comanda
4     public static void main(String[] args)
5     {
6         if (args.length == 0)
7         {
8             System.out.println("Nu exista argumente");
9             return;
10        }
11
12        double suma = 0;
13
14        for (int i = 0; i < args.length; ++i)
15        {
16            suma += Double.parseDouble(args[i]);
17        }
18
19        System.out.println("Suma argumentelor: " + suma);
20    }
21 }
```

Rezumat

Capitolul de față a prezentat tipurile referință. O referință este o variabilă care stochează adresa de memorie unde se află un obiect sau referința specială `null`. Referințele pot indica doar obiecte, nu și variabile de tipuri primitive. Orice obiect poate fi referit prin una sau mai multe variabile referință.

Deoarece în Java există doar opt tipuri primitive, aproape totul se traduce în obiecte și clase. Dintre obiecte, stringurile au un tratament mai special, pentru că se pot folosi pentru concatenare operatorii `+` și `+=`. În rest, stringurile sunt la fel ca orice alt tip referință. Pentru a testa dacă două stringuri sunt egale ca și conținut, se folosește metoda `equals`.

Șirurile reprezintă o colecție de elemente de același tip. Este important să reținem faptul că indicele de numerotare a elementelor din șir pornește întotdeauna de la 0.

În cadrul capitolului care urmează vom prezenta modul în care putem defini noi tipuri de date în Java, folosind noțiunea de *clase*.

Noțiuni fundamentale

apelare prin referință: în majoritatea limbajelor de programare, aceasta înseamnă că parametrul formal reprezintă o referință către parametrul actual. Acest efect este atins în mod natural în Java în momentul în care se utilizează apelul prin valoare pentru tipuri referință.

argument în linie de comandă: argumente transmise la execuția unui program Java și care sunt preluate în funcția `main()`.

colectarea de gunoaie: eliberarea automată din memorie a obiectelor care nu mai sunt referite.

construirea: pentru obiecte, se realizează prin intermediul cuvântului cheie `new`.

`equals`: metodă prin care se poate compara dacă valorile stocate de două obiecte sunt egale.

`length` (**atribut**): folosit pentru a determina dimensiunea unui șir.

`length` (**metodă**): folosită pentru a determina lungimea unui obiect de tip `String`.

obiect: o entitate de tip neprimitiv.

`new`: folosit pentru a construi noi obiecte.

`null`: referință specială prin care se specifică faptul că nu se face referire la nici un obiect.

`NullPointerException`: tip de excepție generată de încercarea de a accesa un atribut sau o metodă pe o referință `null`.

pointer: ca și referința, pointerul conține adresa la care se află un obiect. Spre deosebire de referințe, pointerii necesită o dereferențiere explicită pentru a putea manipula obiectul indicat. În Java nu există pointeri.

referință: variabilă care stochează adresa la care se află un obiect, și prin intermediul căreia poate fi manipulat obiectul care este referit.

șir: reține o colecție de obiecte de același tip

Erori frecvente

1. Pentru tipuri referință operatorul `=` nu copiază valorile obiectelor. El copiază doar adrese.

2. Pentru tipuri referință (inclusiv stringuri) trebuie folosită metoda `equals` în loc de `==` pentru a testa dacă obiectele referite sunt egale ca și conținut.
3. Alocarea cu un element mai puțin decât trebuie pentru șiruri (indexarea începe de la 0!).
4. Tipurile referință sunt implicit inițializate cu `null`. Nici un obiect nu este construit fără apelul lui `new`. O excepție `NullPointerException` indică faptul că ai uitat să alocați memorie pentru un obiect.
5. În Java șirurile sunt indexate la 0 la $N - 1$, unde N este dimensiunea șirului. Totuși Java verifică valoarea indicilor, și accesul în afara limitelor este detectat în timpul execuției.
6. Șirurile bidimensionale sunt indexate prin `A[i][j]` și nu `A[i, j]` ca în Pascal.
7. Folosiți `" "` și nu `' '` pentru a scrie un spațiu sau alte caractere.

Exerciții

Pe scurt

1. Care sunt diferențele majore între tipurile primitive și tipurile referință?
2. Enumerați 5 operații care se pot aplica unui tip referință.
3. Enumerați operațiile de bază care pot fi efectuate pe stringuri.

Teorie

1. Dacă `x` și `y` au valorile 5 respectiv 7, care este rezultatul următoarei afișări:

```
System.out.println(x + ' ' + y);
System.out.println(x + " " + y);
```

În practică

1. Creați un șir de obiecte de tip `String` și asociați un `String` fiecărui element din șir. Afișați apoi elementele șirului în cadrul unui ciclu `for`.
2. Scrieți o metodă care returnează `true` dacă stringul `str1` este prefix pentru stringul `str2`. Nu folosiți nici o metodă generală de căutare pe stringuri în afară de `charAt`.

3. Scrieți un program care preia trei argumente de tip String din linia de comandă și le afișează pe ecran.

4. Obiecte și clase

Cuvintele sau limbajul, așa cum sunt ele scrise sau vorbite nu par să joace nici un rol în mecanismul gândirii mele. Obiectele care par să serverască drept elemente în gândirea mea sunt anumite semne și imagini mai mult sau mai puțin clare care pot fi în mod voluntar reproduse sau combinate.

Albert Einstein

În acest capitol vom începe să discutăm despre programarea orientată pe obiecte (object oriented programming - OOP) în Java. O componentă fundamentală a programării orientate pe obiecte este specificarea, implementarea și folosirea obiectelor. În capitolul anterior am văzut deja câteva exemple de obiecte, cum ar fi stringurile, care fac parte din bibliotecile limbajului Java. Am putut observa și faptul că fiecare obiect este caracterizat de o anumită stare care poate fi modificată prin aplicarea operatorului punct (.). În limbajul Java, starea și funcționalitatea unui obiect se definesc prin intermediul unei clase. Un obiect este de fapt o instanță a unei clase.

În cadrul capitolului de față vom prezenta:

- Cum se folosește în Java conceptul de clasă pentru a obține încapsularea și ascunderea de informații, concepte fundamentale ale OOP;
- Cum se implementează o clasă Java;
- Cum se pot grupa clasele în pachete pe baza funcționalității lor comune.

4.1 Ce este programarea orientată pe obiecte?

Programarea orientată pe obiecte s-a impus ca modelul dominant al anilor '90 și continuă să domine și în deceniul curent. În această secțiune vom prezenta modul în care Java suportă programarea orientată pe obiecte și vom menționa câteva dintre principiile ei fundamentale.

În centrul programării orientate pe obiecte se află noțiunea de *obiect*. Obiectul este o variabilă complexă definit de o *structură* și o *stare*. Fiecare obiect dispune de *operații* prin intermediul cărora i se poate manipula starea. Așa cum am văzut deja, în limbajul Java se face distincție între un obiect și o variabilă de un tip primitiv, dar aceasta este o specificitate a limbajului Java și nu a programării orientate pe obiecte în general. Pe lângă operațiile cu un caracter general, asupra obiectelor se mai pot realiza și alte operații:

- Crearea de noi obiecte, însoțită eventual de inițializarea obiectelor;
- Copierea și testarea egalității;
- Realizarea de operații de intrare/ieșire cu obiecte.

Obiectul trebuie privit ca o unitate *atomică* pe care utilizatorul nu ar trebui să o disece. În mod normal, nu ne punem problema de a jongla cu biții din care este format un număr reprezentat în virgulă mobilă și ar fi de-a dreptul ridicol să încercăm să incrementăm un astfel de număr prin modificarea directă a reprezentării sale interne.

Principiul atomicității este cunoscut sub numele de *ascunderea informației*. Utilizatorul nu are acces direct la componentele unui obiect sau la implementarea sa. Acestea vor putea fi accesate doar prin intermediul *metodelor* care au fost furnizate împreună cu obiectul. Putem privi fiecare obiect ca fiind ambalat într-o cutie pe care este scris "Nu deschideți! Nu conține componente reparabile de către utilizator!". În viața de zi cu zi, majoritatea celor care încearcă să repare componente cu această inscripție sfârșesc prin a face mai mult rău decât bine. Din acest punct de vedere, programarea *imită* lumea reală. Gruparea datelor și a operațiilor asupra acestor date în același întreg (agregat), având grijă să ascundem detaliile de implementare ale agregatului, este cunoscută sub numele de *încapsulare*. Așadar, datele sunt ascunse, iar accesul lor se realizează prin intermediul operațiilor încapsulate împreună cu ele, numite metode.

Unul dintre principalele scopuri ale programării orientate pe obiecte este reutilizarea codului. La fel cum inginerii refolosesc din nou și din nou aceleași componente în proiectarea de componente electronice, programatorii ar trebui

să refolosească obiectele în loc să le reimplementeze. Există două situații distincte legate de reutilizarea codului (refolosirea obiectelor):

1. Situația în care avem deja la dispoziție un obiect care implementează exact comportamentul pe care îl dorim. Refolosirea obiectului nu pune în acest caz nici un fel de probleme;
2. Situația în care dorim să folosim un obiect care deja există, dar care, deși are un comportament foarte similar cu ceea ce vrem, nu corespunde exact cu necesitățile noastre. Aici este de fapt adevărata provocare: obiectul existent va trebui extins pentru a fi adaptat la comportamentul dorit. Astfel se economisește timp de dezvoltare, deoarece adeseori este mult mai ușor să se adapteze un obiect deja existent (a cărui funcționalitate este asemănătoare cu cea a obiectului de care avem nevoie) decât să se rescrie totul de la zero.

Limbajele de programare orientate pe obiecte furnizează mai multe mecanisme pentru a facilita reutilizarea codului. Unul dintre mecanisme este folosirea codului *generic*. Dacă implementarea este identică, și diferă doar tipul de bază al obiectului, nu este necesar să rescriem complet codul: vom scrie în schimb un cod generic care funcționează pentru orice tip. De exemplu, algoritmul de sortare al unui șir de obiecte nu depinde de obiectele care sunt sortate, deci se poate implementa un algoritm generic de sortare.

Moștenirea este un alt mecanism care permite extinderea funcționalității unui obiect. Cu alte cuvinte, putem crea noi tipuri de date care să extindă (sau să restricționeze) proprietățile tipului de date original.

Un alt principiu important al programării orientate pe obiecte este *polimorfismul*. Un tip referință *polimorfic* poate să refere obiecte de mai multe tipuri. Atunci când se apelează o metodă a tipului *polimorfic*, se va selecta automat metoda care corespunde tipului referit în acel moment (vom descrie în detaliu atât moștenirea cât și polimorfismul în capitolul următor).

Un *obiect* în Java este o instanță a unei *clase*. O clasă este similară cu un tip record din Pascal sau cu o structură din C, doar că există două îmbunătățiri majore. În primul rând, membrii clasei pot fi atât funcții cât și date, numite în acest context *metode*, respectiv *atribute*. În al doilea rând, domeniul de vizibilitate al acestor membri poate fi restricționat. Deoarece metodele care manipulează starea obiectului sunt membri ai clasei, ele sunt accesate prin intermediul operatorului punct, la fel ca și atributele. În terminologia programării orientate pe obiecte, atunci când apelăm o metodă a obiectului spunem că "trimitem un mesaj" obiectului.

4.2 Un exemplu simplu

Să ne amintim că, atunci când proiectăm o clasă, este important să ascundem detaliile interne față de utilizatorul clasei. Clasa poate să își definească funcționalitatea prin intermediul *metodelor*. Unele dintre aceste metode vor descrie cum se creează și se inițializează o instanță a clasei, cum se realizează testele de egalitate și cum se descrie starea clasei. Celelalte metode sunt specifice structurii particulare pe care o are clasa. Ideea este că utilizatorul nu trebuie să aibă dreptul de a modifica direct starea obiectului, ci el va trebui să folosească metodele clasei pentru a realiza acest lucru. Această idee poate fi impusă prin ascunderea anumitor membri față de utilizator. Pentru a realiza aceasta, vom preciza ca acești membri să fie stocați în secțiunea `private`. Compilatorul va avea grijă ca membri din secțiunea `private` să fie inaccesibili utilizatorului acelui obiect. În general, toate atributele unui obiect ar trebui să fie declarate `private`.

Programul din **Listing 4.1** prezintă modul de definire al unei clase care modelează un cerc. Definirea clasei constă în două părți: `public` și `private`. Secțiunea `public` reprezintă porțiunea care este vizibilă pentru utilizatorul obiectului. Deoarece datele sunt ascunse față de utilizator, secțiunea `public` va conține în mod normal numai metode și constante. În exemplul nostru avem două metode, una pentru a scrie și una pentru a citi raza obiectelor de tip `Circle`. Celelalte două metode calculează aria respectiv lungimea obiectului de tip `Circle`. Secțiunea `private` conține datele; acestea sunt invizibile pentru utilizatorul obiectului. Atributul `radius` poate fi accesat doar prin intermediul metodelor publice `setRadius()` și `getRadius()`.

Listing 4.1: Definirea clasei `Circle`

```

1 /**
2  * Clasa simpla Java care modeleaza un Cerc.
3  */
4  public class Circle
5  {
6      //raza cercului
7      //valoarea razei nu poate fi modificata
8      //direct de catre utilizator
9      private double radius;
10
11     /** Modifica raza cercului */
12     public void setRadius(double r)
13     {
14         radius = r;
15     }
16     /** Metoda ptr a obtine raza cercului. */

```

```

17 public double getRadius ()
18 {
19     return radius;
20 }
21 /** Metoda ptr calculul ariei cercului.*/
22 public double area ()
23 {
24     return Math.PI * radius * radius;
25 }
26 /** Metoda ptr calculul lungimii.*/
27 public double length ()
28 {
29     return 2 * Math.PI * radius;
30 }
31 }

```

Programul din **Listing 4.2** prezintă modul de folosire al unui obiect de tip `Circle`. Deoarece `setRadius()`, `getRadius()`, `area()` și `length()` sunt membri ai clasei, ei sunt accesați folosind operatorul punct. Atributul `radius` ar fi putut și el să fie accesat folosind operatorul punct, dacă nu ar fi fost declarat de tip `private`. Accesarea lui `radius` din linia 15 ar fi fost ilegală dacă nu era comentată folosind `//`.

Să rezumăm terminologia învățată. O clasă conține *membri* care pot fi *atribute* (câmpuri, date) sau *metode* (funcții). Metodele pot acționa asupra atributelor și pot apela alte metode. Modificatorul de vizibilitate `public` face ca membrul respectiv să fie accesibil oricui prin intermediul operatorului punct. Modificatorul de vizibilitate `private` face ca membrul respectiv să fie accesibil doar metodelor clasei. Dacă nu se pune nici un modificator de vizibilitate, atunci accesul la membru este de tip *friendly*, despre care vom vorbi în paragraful 4.4.4. Mai există și un al patrulea modificator, numit `protected` pe care îl vom prezenta în capitolul următor.

Listing 4.2: Testarea clasei `Circle`

```

1 //clasa simpla de testare a clasei Circle
2 public class TestCircle
3 {
4     public static void main(String[] args)
5     {
6         Circle circle = new Circle();
7
8         circle.setRadius(10);
9         System.out.println("Raza=" + circle.getRadius());
10        System.out.println("Aria=" + circle.area());
11        System.out.println("Lungimea=" + circle.length());
12
13        //urmatoarea linie ar genera o

```

```
14      //eroare de compilare
15      //circle.radius = 20;
16    }
17 }
```

4.3 Metode uzuale

Metodele unei clase se pot grupa în două categorii:

- metode clasice, uzuale, care se regăsesc în (aproape) toate clasele;
- metode care definesc comportamente specifice unei anumite clase.

În această secțiune vom prezenta prima categorie de metode: constructorii, modificatorii, accesorii, `toString()` și `equals()`.

4.3.1 Constructori

Așa cum am menționat deja, una dintre proprietățile fundamentale ale obiectelor este că acestea pot fi definite și, eventual, inițializate. În limbajul Java, metoda care controlează modul în care un obiect este creat și inițializat este *constructorul*. Deoarece Java permite supraîncărcarea metodelor, o clasă poate să definească mai mulți constructori.

Dacă la definirea clasei nu se furnizează nici un constructor, cum este cazul clasei `Circle` din **Listing 4.1**, compilatorul creează automat un constructor implicit care inițializează fiecare membru cu valorile implicite. Aceasta înseamnă că atributele de tipuri primitive sunt inițializate cu 0 (cele boolean cu `false`), iar atributele de tip referință sunt inițializate cu `null`. Astfel, în cazul nostru, atributul `radius` va avea implicit valoarea 0.

Pentru a furniza un constructor, vom scrie o metodă care are același nume cu clasa și care nu returnează nimic. În **Listing 4.3** avem doi constructori: unul începe la linia 7, iar celălalt la linia 15. Folosind acești doi constructori vom putea crea obiecte de tip `Date` în următoarele moduri:

```
Date d1 = new Date();
Date d2 = new Date(15, 3, 2000);
```

De remarcat faptul că odată ce ai definit un constructor pentru o clasă, compilatorul nu mai generează constructorul implicit fără parametri. Dacă veți avea nevoie de un constructor fără parametri, va trebui acum să îl scrieți singuri. De exemplu, constructorul din linia 7 trebuie definit obligatoriu pentru a putea crea un obiect de tipul celui referit de către `d1`.

Listing 4.3: O clasă Date minimală care ilustrează constructorii și metodele equals() și toString()

```

1  //clasa Java simpla pentru stocarea unei
2  //date calendaristice
3  //nu se face validarea datelor
4  public class Date
5  {
6      //constructor fara parametri
7      public Date()
8      {
9          day = 1;
10         month = 1;
11         year = 2000;
12     }
13
14     //constructor cu trei parametri
15     public Date(int theDay, int theMonth, int theYear)
16     {
17         day = theDay;
18         month = theMonth;
19         year = theYear;
20     }
21
22     //test de egalitate
23     //intoarce true daca Obiectul x
24     //este egal cu obiectul curent
25     public boolean equals(Object x)
26     {
27         if (!(x instanceof Date))
28             return false;
29         Date date = (Date) x;
30         return date.day == day && date.month == month
31             && date.year == year;
32     }
33
34     //conversie la String
35     public String toString()
36     {
37         return day + "/" + month + "/" + year ;
38     }
39
40     //atribute
41     private int day;
42     private int month;
43     private int year;
44 }

```

4.3.2 Modificatori și accesorii

Atributele sunt declarate de obicei ca fiind *private*. Aceasta înseamnă că ele nu vor putea fi direct accesate de către rutinele care nu aparțin clasei. Există totuși multe situații în care dorim să examinăm sau chiar să modificăm valoarea unui atribut.

O posibilitate este aceea de a declara atributele ca fiind *public*. Aceasta nu este o soluție elegantă, deoarece încalcă principiul ascunderii informației. Putem însă scrie metode care să examineze sau să modifice valoarea fiecărui câmp. O metodă care citește, dar nu modifică starea unui obiect este numită *accesor*. O metodă care modifică starea unui obiect este numită *modificator* (engl. *mutator*).

Cazuri particulare de accesorii și modificatori sunt cele care acționează asupra unui singur câmp. Accesorii de acest tip au un nume care începe de obicei cu `get`, cum ar fi `getRadius()`, iar modificatorii au un nume care începe de regulă cu `set`, cum ar fi `setRadius()`. Aceste metode sunt atât de des folosite în Java, încât au și denumiri consacrate în limbajul uzual al programatorilor Java: *getteri*, respectiv *setteri* (a nu se confunda cu rasa de patrupede care poartă același nume).

Avantajul folosirii unui modificator (sau a unui setter) este că acesta poate verifica dacă starea obiectului este corectă. Astfel, un setter care modifică atributul `day` al unui obiect de tip `Date` poate verifica corectitudinea datei care rezultă.

4.3.3 Afișare și `toString()`

În general, afișarea stării unui obiect se face utilizând metoda `print()` din clasa `System.out`. Pentru a putea face acest lucru trebuie ca obiectul care se dorește a fi afișat să conțină o metodă cu numele de `toString()`. Această metodă întoarce un `String` (reprezentând starea obiectului) care poate fi afișat. Ca un exemplu, în **Listing 4.3** am prezentat o implementare rudimentară a unei metode `toString()` pentru clasa `Date` în liniile 35-38. Definirea acestei metode permite să afișăm un obiect `d1` de tip `Date` folosind instrucțiunea `System.out.print(d1)`. Practic, compilatorul Java nu face altceva decât să invoce automat `toString()` pentru fiecare obiect care se afișează. Astfel `System.out.print(d1)` este tradusă de către compilator în instrucțiunea `System.out.print(d1.toString())`. Această simplă facilități lasă impresia programatorilor neavizați că metoda `print()` “știe” să afișeze obiecte Java. În realitate, metoda `print()` nu știe decât să afișeze stringuri. Compilatorul Java este însă suficient de inteligent pentru a transforma obiectele

în stringuri, prin apelul metodei `toString()` (de altfel, tot compilatorul este cel care convertește la afișare și tipurile primitive la `String`, atunci când este cazul).

4.3.4 Metoda `equals()`

Metoda `equals` este folosită pentru a testa dacă două referințe indică obiecte care au aceeași valoare (stare). Antetul acestei metode este întotdeauna

```
public boolean equals(Object rhs)
```

Ați remarcat probabil nedumeriți faptul că parametrul trimis metodei este de tip `Object` și nu de tip `Date`, cum ar fi fost de așteptat. Rațiunea acestui lucru o să o prezentăm în capitolul următor. În general, metoda `equals` pentru o clasă `X` este implementată în așa fel încât să returneze `true` doar dacă `rhs` (abreviere pentru "right hand side", adică operandul din partea dreaptă a expresiei) este o instanță a lui `X` și, în plus, după conversia la `X` toate atributele lui `rhs` sunt egale cu atributele obiectului (atributele de tip primitiv trebuie să fie egale via `==`, iar atributele de tip referință trebuie să fie egale via `equals()`).

Un exemplu de implementare a lui `equals` este dat în **Listing 4.3** pentru clasa `Date` în liniile 25-32.

4.3.5 Metode statice

Există anumite cuvinte cheie ale limbajului Java care specifică proprietăți speciale pentru unele *atribute* sau *metode*. Un astfel de cuvânt cheie este `static`. Atributele și metodele declarate `static` într-o clasă, sunt aceleași pentru toate obiectele, adică pentru toate variabilele de tipul acelei clase. Fiind identice pentru toate obiectele unei clase, metodele statice pot fi accesate fără să fie nevoie de o *instanțiere* a clasei respective (adică de o variabilă de clasa respectivă). Metodele statice pot utiliza variabile statice declarate în interiorul clasei.

Cel mai cunoscut exemplu de metodă statică este `main()`. Alte exemple de metode statice pot fi găsite în clasele `String`, `Integer` și `Math`. Exemple de astfel de metode sunt `String.valueOf()`, `Integer.parseInt()`, `Math.sin()` și `Math.max()`. Accesul la metodele statice respectă aceleași reguli de vizibilitate ca și metodele normale.

4.3.6 Atribute statice

Atributele statice sunt folosite în situația în care avem variabile care trebuie partajate de către toate instanțele unei clase. De obicei atributele statice sunt

constante simbolice, dar acest lucru nu este obligatoriu. Atunci când un atribut al unei clase este declarat de tip static, doar o singură instanță a acelei variabile va fi creată. Ea nu face parte din nici o instanță a clasei. Ea se comportă ca un fel de variabilă globală unică, vizibilă în cadrul clasei. Cu alte cuvinte, dacă avem declarația

```
1 public class Exemplu
2 {
3     private int x;
4     private static int y;
5 }
```

fiecare obiect de tip `Exemplu` va avea propriul atribut `x`, dar va fi doar un singur `y` partajat de toate instanțele clasei `Exemplu`.

O folosire frecventă pentru câmpurile statice o reprezintă constantele. De exemplu, clasa `Integer` definește atributul `MAX_VALUE` astfel

```
public final static int MAX_VALUE = 2147483647;
```

Analog se definește și constanta `PI` din clasa `Math` pe care am folosit-o în clasa `Circle`. Modificatorul `final` indică faptul că identificatorul care urmează este o constantă. Dacă această constantă nu ar fi fost un atribut static, atunci fiecare instanță a clasei `Integer` ar fi avut un atribut cu numele de `MAX_VALUE`, irosind astfel inutil spațiu în memorie.

Astfel, vom avea o singură variabilă cu numele de `MAX_VALUE`. Constanta poate fi accesată de oricare dintre metodele clasei `Integer` prin identificatorul `MAX_VALUE`. Ea va putea fi folosită și de către un obiect de tip `Integer` numit `x` prin sintaxa `x.MAX_VALUE`, ca orice alt câmp. Acest lucru este permis doar pentru că `MAX_VALUE` este public. În sfârșit, `MAX_VALUE` poate fi folosit și prin intermediul numelui clasei ca `Integer.MAX_VALUE` (tot pentru că este public). Această ultimă folosire nu ar fi fost permisă pentru un câmp care nu este static.

Chiar și fără modificatorul `final`, atributele statice sunt foarte folositoare. Să presupunem că vrem să reținem numărul de obiecte de tip `Circle` care au fost construite. Pentru aceasta avem nevoie de o variabilă statică. În clasa `Circle` vom face declarația:

```
private static int numarInstante = 0;
```

Vom putea apoi incrementa numărul de instanțe în constructor. Dacă acest câmp nu ar fi fost de tip static am avea un comportament incorect, deoarece fiecare obiect de tip `Circle` ar fi avut propriul atribut `numarInstante` care ar fi fost incrementat de la 0 la 1.

Remarcați faptul că, deoarece un atribut de tip static nu necesită un obiect care să îl controleze, fiind partajat de către toate instanțele clasei, el poate fi

folosit de către o metodă statică (dacă regulile de vizibilitate permit acest lucru). Atributele nestatice ale unei clase vor putea fi folosite de către o metodă statică doar dacă se furnizează și un obiect care să le controleze.

4.3.7 Metoda `main()`

Atunci când este invocat, interpretorul `java` caută metoda `main()` din clasa care i se dă ca parametru. Astfel, în situația în care aplicația noastră este formată din mai multe clase, putem scrie câte o metodă `main()` pentru fiecare clasă. Acest lucru ne permite să testăm funcționalitatea de bază a claselor individuale. Trebuie să avem totuși în vedere faptul că plasarea funcției `main()` în cadrul clasei ne conferă mai multă vizibilitate decât ne-ar fi permis prin utilizarea clasei din alt loc. Astfel, apeluri ale metodelor `private` pot fi făcute în test, dar ele vor eșua când vor fi utilizate în afara clasei.

4.4 Pachete

Pachetele sunt folosite pentru a organiza clasele similare. Fiecare pachet constă dintr-o mulțime de clase. Clasele care sunt în același pachet au restricții de vizibilitate mai slabe între ele decât clasele din pachete diferite.

Java oferă o serie de pachete predefinite, printre care `java.io`, `java.awt`, `java.lang`, `java.util`, `java.applet` etc. Pachetul `java.lang` include, printre altele, clasele `Integer`, `Math`, `String` și `System`. Clase mai cunoscute din `java.util` sunt `Date`, `Random`, `StringTokenizer`. Pachetul `java.io` cuprinde diverse clase pentru stream-uri, pe care le vom prezenta în capitolul 7. De asemenea, pentru o prezentare mai amănunțită a pachetelor predefinite, vă recomandăm să consultați anexa C. Pe lângă informații detaliate despre aceste pachete și despre modalitatea de a vă crea propriile dumneavoastră pachete, anexa C prezintă și noțiunea de arhivă *jar*, foarte utilă în cadrul aplicațiilor de dimensiuni mai mari.

O clasă oarecare `C` dintr-un pachet `P` este desemnată prin `P.C`. De exemplu, putem declara un obiect de tip `Date` care să conțină data și ora curentă astfel:

```
java.util.Date today = new java.util.Date();
```

Observați că prin specificarea numelui pachetului din care face parte clasa, evităm conflictele care pot fi generate de clase cu același nume din pachete diferite¹.

¹De exemplu, cu clasa `Date` definită anterior în secțiunea 4.3.

4.4.1 Directiva import

Utilizarea permanentă a numelui pachetului din care fac parte clasele poate fi uneori deosebit de anevoioasă. Pentru a evita acest lucru se folosește directiva `import`:

```
import NumePachet . NumeClasa ;
```

sau

```
import NumePachet . * ;
```

Dacă recurgem la prima formă a directivei `import`, vom putea folosi denumirea `NumeClasa` ca o prescurtare pentru numele clasei cu calificare completă. Dacă folosim cea de-a doua formă, toate clasele din pachet vor putea fi abreviate cu numele lor neprecedat de numele pachetului.

De exemplu, realizând directivele `import` de mai jos:

```
1 import java . util . Date ;
2 import java . io . * ;
```

putem să folosim:

```
1 Date today = new Date () ;
2 FileReader file = new FileReader (name) ;
```

Folosirea directivelor `import` economisește timpul de scriere. Având în vedere că cea de-a doua formă este mai generală, ea este și cel mai des folosită. Există două dezavantaje ale folosirii directivei `import`. Primul dezavantaj este acela că la citirea codului va fi mai greu de stabilit pachetul din care o anumită clasă face parte. Al doilea este acela că folosirea celei de-a doua forme poate să introducă prescurtări neintenționate pentru anumite clase, ceea ce va genera conflicte de denumire care vor trebui rezolvate prin folosirea de nume de clase calificate.

Să presupunem că avem următoarele directive:

```
1 import java . util . * ; // pachet predefinit
2 import myutil . * ; // pachet definit de catre noi
```

cu intenția de a importa clasa `java.util.Random` și o clasă pe care am definit-o chiar noi. Atunci, dacă noi avem propria clasă `Date` în pachetul `myutil`, directiva `import` va genera un conflict cu `java.util.Date` și, din această cauză, clasa va trebui să fie complet calificată ori de câte ori este utilizată. Am fi putut evita aceste probleme dacă am fi folosit prima formă

```
import java . util . Random ;
```

Directivele `import` trebuie să apară înainte de orice declarație a unei clase. Pachetul `java.lang` este automat inclus în întregime. Acesta este motivul pentru care putem folosi prescurtări de genul `Integer.parseInt()`, `System.out`, `Math.max()` etc.

4.4.2 Instrucțiunea `package`

Pentru a indica faptul că o clasă face parte dintr-un anumit pachet trebuie să realizăm două lucruri. În primul rând trebuie să scriem o instrucțiune `package` pe prima linie, înainte de a declara clasa. Apoi, va trebui să plasăm clasa în directorul corespunzător. Mai multe detalii veți găsi în cadrul secțiunii C.2 a anexei C.

4.4.3 Variabila sistem `CLASSPATH`

Compilatorul și interpretorul Java caută automat pachetele și clasele Java în directoarele care sunt precizate în variabila sistem `CLASSPATH`. Ce înseamnă aceasta? Iată o posibilă setare pentru `CLASSPATH`, mai întâi pentru un sistem de operare Windows 95/98/2000, iar apoi pentru un sistem Unix/Linux:

```
SET CLASSPATH=.;c:\jdk1.4\lib\
export CLASSPATH=./usr/local/jdk1.4/lib:$HOME/
                                javawork
```

În ambele cazuri variabila `CLASSPATH` conține directoarele (sau arhivele) care conțin programele compilate având extensia `.class`. De exemplu, dacă variabila `CLASSPATH` nu este setată corect, nu veți putea compila nici măcar cel mai banal program Java, deoarece pachetul `java.lang` nu va fi găsit. O clasă care se află în pachetul `P` va trebui să fie pusă într-un director cu numele `P` care să se afle într-un director din `CLASSPATH`. Directorul curent (`.`) este întotdeauna în variabila `CLASSPATH`, deci dacă lucrați într-un singur director principal, puteți crea subdirectoarele chiar în el. Totuși, în majoritatea situațiilor, veți dori să creați un subdirector Java separat și să creați directoarele pentru pachete în acel subdirector. În această situație va trebui să adăugați la variabila `CLASSPATH` acest director. Acest lucru a fost realizat în exemplul de mai sus prin adăugarea directorului `$HOME/javawork` la `CLASSPATH` (notația `$HOME`, familiară utilizatorilor de Linux, reprezintă în acest context un alias pentru un director propriu-zis; de exemplu, `/home/george`). Utilizatorii de Windows pot adapta la fel de ușor variabila sistem `CLASSPATH` conform necesităților lor. Odată aceste modificări încheiate, în directorul `javawork` veți putea crea subdirectorul `io`. În subdirectorul `io` vom plasa codul pentru clasa

Reader. Clasa Reader este o clasă ajutătoare, care ne permite să citim foarte comod, variabile de tip `int`, `char`, `String`, `float` și chiar șiruri de numere întregi. Exemplele din volumul al doilea al cărții vor utiliza intensiv această clasă utilitară.

Codul sursă al clasei este prezentat în **Listing 4.4** (detalii despre instrucțiunea `try-catch` vom prezenta în capitolul 6, iar despre clasele de intrare/ieșire în capitolul 7).

O aplicație va putea în această situație să folosească metoda `readInt()` fie prin

```
int x = io.Reader.readInt();
```

sau pur și simplu prin

```
int x = Reader.readInt();
```

dacă se furnizează directiva `import` corespunzătoare.

Listing 4.4: Clasa Reader

```

1 package io;
2 //clasa va trebui salvata intr-un director cu numele "io"
3 //directorul in care se afla "io" va trebui adaugat
4 //in CLASSPATH
5 import java.io.*;
6 import java.util.StringTokenizer;
7 public class Reader
8 {
9     public static String readString()
10    {
11        BufferedReader in = new BufferedReader(
12            new InputStreamReader(System.in));
13        try
14        {
15            return in.readLine();
16        }
17        catch(IOException e)
18        {
19            //ignore
20        }
21        return null;
22    }
23
24    public static int readInt()
25    {
26        return Integer.parseInt(readString());
27    }
28
29    public static double readDouble()
30    {

```

```

31     return Double.parseDouble(readString());
32 }
33
34 public static char readChar()
35 {
36     BufferedReader in = new BufferedReader(
37         new InputStreamReader(System.in));
38     try
39     {
40         return (char)in.read();
41     }
42     catch (IOException e)
43     {
44         //ignore
45     }
46     return '\0';
47 }
48
49 public static int[] readIntArray()
50 {
51     String s = readString();
52     StringTokenizer st = new StringTokenizer(s);
53     //aloca memorie pentru sir
54     int[] a = new int[st.countTokens()];
55
56     for (int i = 0; i < a.length; ++i)
57     {
58         a[i] = Integer.parseInt(st.nextToken());
59     }
60
61     return a;
62 }
63
64 }

```

4.4.4 Reguli de vizibilitate package-friendly

Pachetele au câteva reguli de vizibilitate importante. În primul rând, dacă pentru un membru al unei clase nu se precizează nici un modificador de vizibilitate (`public`, `protected` sau `private`), atunci membrul respectiv devine (package) *friendly*. Aceasta înseamnă că acel câmp este vizibil doar pentru clasele din cadrul aceluiași pachet. Aceasta este o vizibilitate mai puțin restrictivă decât `private`, dar mai restrictivă decât `public` (care este vizibil și pentru membrii din alte clase, chiar și din pachete diferite).

În al doilea rând, doar clasele `public` din cadrul unui pachet pot fi folosite din afara pachetului. Acesta este motivul pentru care am pus întotdeauna modi-

ficatorul public în fața unei clase. Clasele nu pot fi declarate private sau *protected* (cu o singură excepție: clasele interioare, despre care vom vorbi în capitolul următor). Accesul de tip *friendly* se extinde și pentru clase. Dacă o clasă nu este declarată ca fiind de tip *public*, atunci ea va putea fi accesată doar de clasele din cadrul aceluiași pachet.

Toate clasele care nu fac parte din nici un pachet, dar sunt accesibile fiind puse într-un director din `CLASSPATH`, sunt considerate automat ca făcând parte din același pachet implicit. Ca o consecință, accesul de tip *friendly* se aplică pentru toate aceste clase. Acesta este motivul pentru care vizibilitatea nu este afectată dacă ometem să punem modificatorul *public* în clasele care nu fac parte dintr-un pachet. Totuși, această modalitate de folosire a accesului *friendly* nu este recomandată.

4.4.5 Compilarea separată

Atunci când o aplicație constă din mai multe fișiere sursă `.java`, fiecare fișier trebuie compilat separat. În mod normal, fiecare clasă este plasată într-un fișier `.java` propriu. Ca urmare a compilării vom obține o colecție de fișiere `.class`. Clasele sursă pot fi compilate în orice ordine. Pentru a compila toate fișierele sursă dintr-un director printr-o singură comandă, puteți folosi o extensie a comenzii `javac`, astfel:

```
javac *.java
```

4.5 Alte operații cu obiecte și clase

În această secțiune vom prezenta încă trei cuvinte cheie importante: `this`, `instanceof` și `static`. `this` are mai multe utilizări în Java. Două dintre ele le vom prezenta în această secțiune. `instanceof` are și el mai multe utilizări; îl vom folosi aici pentru a ne asigura că o conversie de tip se poate realiza. Și cuvântul cheie `static` are mai multe semnificații. Vom vorbi despre metode statice, atribute statice și inițializatori statici.

4.5.1 Referința `this`

Cea mai cunoscută utilizare pentru `this` este ca o referință la obiectul curent. Imaginați-vă că `this` vă indică în fiecare moment locul unde vă aflați. O utilizare tipică pentru `this` este calificarea atributelor unei clase în cadrul

unei metode a clasei care primește parametri cu nume identic cu numele atributelor. De exemplu, în clasa `Circle`, din **Listing 4.1**, putem defini metoda `setRadius` astfel:

```
1 /** Modifica raza cercului. */
2 public void setRadius(double radius)
3 {
4     //radius este parametrul,
5     //iar this.radius este atributul clasei
6     this.radius = radius;
7 }
```

În secvența de cod precedentă, pentru a face distincție între parametrul `radius` și atributul cu același nume (care este "ascuns" de către parametru) se folosește sintaxa `this.radius` pentru a referi atributul clasei.

Un alt exemplu de folosire al lui `this` este pentru a testa că parametrul pe care o metodă îl primește nu este chiar obiectul curent. Să presupunem, de exemplu, că avem o clasă `Account` care are o metodă `finalTransfer()` pentru a transfera toată suma de bani dintr-un cont în altul. Metoda ar putea fi scrisă astfel:

```
1 public void finalTransfer(Account account)
2 {
3     dollars += account.dollars;
4     account.dollars = 0;
5 }
```

Să considerăm secvența de cod de mai jos:

```
1 Account account1;
2 Account account2;
3 ....
4 account2 = account1;
5 account1.finalTransfer(account2);
```

Deoarece transferăm bani în cadrul aceluiași cont, nu ar trebui să fie nici o modificare în cadrul contului. Totuși, ultima linie din `finalTransfer()` are ca efect golirea contului debitor, ceea ce va face ca suma din `account2` să fie nulă. O modalitate de a evita o astfel de situație este folosirea unui test pentru pseudonime (referințe care indică același obiect):

```
1 public void finalTransfer(Account account)
2 {
3     //daca se incearca un transfer in acelasi cont
4     if (this == account)
5     {
6         return;
7     }
8 }
```

```
9     dollars += account.dollars;
10    account.dollars = 0;
11 }
```

4.5.2 Prescurtarea `this` pentru constructori

Multe clase dispun de mai mulți constructori care au un comportament similar. Putem folosi `this` în cadrul unui constructor pentru a apela ceilalți constructori ai clasei. De exemplu, o altă posibilitate de a scrie constructorul fără parametri pentru clasa `Date` este:

```
1 public Date()
2 {
3     //apeleaza constructorul Date(int, int, int)
4     this(1, 1, 2000);
5 }
```

Se pot realiza și exemple mai complicate, dar întotdeauna apelul lui `this` trebuie să fie prima instrucțiune din constructor, celelalte instrucțiuni fiind în continuarea acesteia.

4.5.3 Operatorul `instanceof`

Operatorul `instanceof` realizează o testare de tip în timpul execuției. Rezultatul expresiei:

```
exp instanceof NumeClasa
```

este `true` dacă `exp` este o instanță a lui `NumeClasa` și `false` în caz contrar. Dacă `exp` este `null`, rezultatul este întotdeauna `false`. `instanceof` este folosit de obicei înainte de o conversie de tip, și adeseori este folosit în legătură cu referințele polimorfe pe care le vom prezenta în capitoul următor.

4.5.4 Inițializatori statici

Atributele statice sunt inițializate atunci când clasa este încărcată în memorie. Uneori este însă nevoie de o inițializare mai complexă a acestor atribute. Să presupunem de exemplu că avem nevoie de un șir static care specifică pentru fiecare număr între 0 și 100 dacă este număr prim sau nu. O posibilitate ar fi să definim o metodă statică și să cerem programatorului să apeleze acea metodă înainte de a folosi șirul.

O alternativă mai elegantă la această soluție este folosirea *inițializatorului static*. Instrucțiunile din cadrul inițializatorului static sunt executate automat la încărcarea clasei în memorie (deci înainte de a fi creată prima instanță), ceea

ce ne asigură că atributele statice sunt corect inițializate la utilizarea clasei. Un exemplu este prezentat în **Listing 4.5**. Aici inițializatorul static se extinde de la linia 5 la linia 20. Inițializatorul static trebuie să urmeze imediat după membrul static.

Listing 4.5: Clasa `Prime`, care folosește o secvență statică de inițializare

```
1 public class Prime
2 {
3     private static boolean prime[] = new boolean[100];
4
5     static
6     {
7         for (int i = 2; i < prime.length; ++i)
8         {
9             prime[i] = true;
10            for (int j = 2; j <= i/2; ++j)
11            {
12                if (i % j == 0)
13                {
14                    prime[i] = false;
15                    break;
16                }
17            }
18        }
19    }
20 }
21 // restul clasei
22 }
```

Rezumat

Acest capitol descrie noțiunile de obiect și clasă, precum și elementele fundamentale legate de acestea, cum ar fi atribute, metode, constructori, reguli de vizibilitate și pachete. Clasa este mecanismul prin care Java permite crearea de noi tipuri referință. Pachetele sunt folosite pentru a grupa clase cu comportament similar. Pentru o mai bună înțelegere a pachetelor și a modului de operare cu ele vă invităm să consultați anexa C. Tot acolo veți afla că toate clasele care compun o aplicație pot fi grupate într-un singur fișier, indiferent dacă sunt organizate pe pachete sau nu. Rezultatul va fi o arhivă `jar`.

Capitolul următor prezintă alte concepte importante ale programării orientate pe obiecte: moștenirea, polimorfismul, programarea generică.

Noțiuni fundamentale

clasă: concept care grupează atribute și metode care sunt aplicate instanțelor clasei.

CLASSPATH: variabilă sistem care specifică directoarele și fișierele arhive în care mașina virtuală JVM caută clasele folosite de aplicațiile Java.

constructor: metodă specială responsabilă cu crearea și inițializarea de noi instanțe ale clasei.

constructor `this`: folosit pentru a apela un alt constructor din aceeași clasă.

friendly: tip de acces prin care clasele/metodele/atributele asociate nu sunt accesibile decât în interiorul pachetului din care fac parte.

import: instrucțiune prin care se oferă o prescurtare pentru un nume complet de clasă (numele complet include și pachetul din care provine clasa).

inițializator static: secvență din cadrul unei clase care permite inițializarea atributelor statice.

obiect: instanță a unei clase.

pachet: termen folosit pentru a organiza colecții de clase cu comportament similar.

package: instrucțiune care indică faptul că o clasă este membră a unui pachet. Trebuie să precedă definirea unei clase.

private: tip de acces prin care atributele/metodele sunt complet ascunse celorlalte clase, putând fi utilizate doar în cadrul clasei din care fac parte.

public: tip de acces prin care atributele/metodele unei clase sunt vizibile în restul claselor.

referința `this`: referință către obiectul curent.

static: cuvânt cheie prin care se precizează faptul că o metodă sau un atribut sunt comune pentru toate instanțele clasei. Membrii statici nu necesită o instanță pentru a fi accesați.

Erori frecvente

1. Membrii `private` nu pot fi accesați din afara clasei. Rețineți că, implicit, membrii unei clase sunt `friendly`. Ei sunt vizibili doar în cadrul pachetului.
2. Folosiți `public class` în loc de `class`, cu excepția situației în care scrieți o clasă ajutătoare de care nu veți mai avea nevoie în afara pachetului.

3. Parametrul formal al metodei `equals` trebuie să fie de tip `Object`. În caz contrar, deși programul se va compila, în anumite situații se va apela metoda `equals` implicită (care compară obiectele folosind `==`).
4. Metodele statice nu pot accesa atributele nestatice fără un obiect care să le controleze.
5. Deseori se uită adăugarea `" . * "` atunci când se dorește importarea tuturor claselor dintr-un pachet.
6. Clasele care fac parte dintr-un pachet trebuie să se afle într-un director cu același nume ca cel al pachetului și care să poate fi "accesat" din cadrul variabilei de sistem `CLASSPATH`.

Exerciții

Pe scurt

1. Ce înseamnă ascunderea informației? Ce este încapsularea? Cum suportă Java aceste concepte?
2. Explicați secțiunile `public` și `private` ale unei clase.
3. Descrieți rolul unui constructor.
4. Dacă o clasă nu are nici un constructor, ce va face compilatorul?
5. Ce înseamnă accesul de tip "package friendly"?

Teorie

1. Să presupunem că funcția `main` din **Listing 4.2**, ar fi făcut parte din clasa `Circle`.
 - Ar fi funcționat programul?
 - Am fi putut comenta linia 15 din metoda `main()` fără a genera erori?

În practică

1. Creați o clasă cu un constructor implicit (fără parametri) care afișează un mesaj. În cadrul metodei `main()`, creați o instanță a clasei și verificați dacă mesajul a fost afișat.

2. Adăugați clasei precedente un constructor care primește un parametru de tip `String`, și care afișează parametrul primit alături de mesajul anterior. Creați câte o instanță a clasei folosind ambii constructori și verificați mesajele afișate.
3. Creați un șir de obiecte de tipul celor de la exercițiul anterior, fără a crea însă obiectele pe care le referă elementele șirului. Verificați dacă mesajele din constructor sunt afișate. Încercați acum să creați câte un obiect pentru fiecare element din șir și observați diferențele.
4. Definiți o clasă fără nici un constructor iar apoi în `main()` creați o instanță a clasei respective pentru a verifica faptul că se creează automat un constructor implicit. Adăugați apoi clasei un constructor cu un parametru de tip `int`, lasând metoda `main()` nemodificată. Programul nu se va mai compila. De ce?
5. Creați o clasă fără nici un constructor care să aibă un atribut de tip `int`, unul de tip `float`, unul de tip `boolean` și unul de tip `String`. Creați o instanță a clasei și afișați valorile acestor atribute pentru a verifica inițializările implicite.
6. Un lacăt cu cifru are următoarele proprietăți: combinația cifrelor (o secvență de trei numere) este ascunsă; lacătul poate fi deschis prin furnizarea combinației corecte; combinația poate fi modificată, dar numai de către cineva care cunoaște combinația corectă. Proiectați o clasă care să definească metodele publice `open()` și `changeCombo()` și câmpuri de tipul `private` care să rețină combinația. Combinația va trebui stabilită în constructor.
7. Scrieți o metodă care crează și inițializează o matrice de `int` cu valori aleatoare. Dimensiunile matricei, precum și intervalul în care pot lua valori elementele matricei sunt date ca parametru. Adăugați o a doua metodă care afișează pe ecran matricea linie cu linie. Verificați cele două metode în `main()` inițializând și afișând matrice de diverse dimensiuni. Încercați apoi să extindeți clasa pentru matrice cu trei dimensiuni.
8. Creați o clasă care să conțină membri de tip `private`, `public` și `friendly`. Creați apoi în cadrul altei clase o instanță a clasei și observați mesajele de eroare pe care le obțineți de la compilator încercând accesarea diversilor membri. Țineți cont de faptul că clasele care nu sunt declarate ca făcând parte dintr-un anumit pachet, fac toate parte din același pachet implicit. Încercați acum să plasați cele două clase în pachete

diferite și observați diferențele între mesajele de eroare afișate la compilare.

Proiecte de programare

1. Scrieți o clasă care suportă numere raționale. Atributele clasei ar trebui să fie două variabile de tip `long`, una pentru numitor și alta pentru numărător. Stocați numărul sub formă de fracție ireductibilă, cu numitorul pozitiv. Furnizați un număr rezonabil de constructori, precum și metodele `add()`, `subtract()`, `multiply()`, `divide()`; de asemenea, metodele `toString()`, `equals()` și `compareTo()` (care se comportă ca cea din clasa `String`). Asigurați-vă că `toString()` funcționează și în cazul în care numitorul este zero.
2. Implementați o clasă pentru numere complexe, `Complex`. Adăugați aceleași metode ca și la clasa `Rational`, dacă au sens (de exemplu, metoda `compareTo()` nu are sens aici). Adăugați accesori pentru partea reală și cea imaginară.
3. Implementați o clasă completă `IntType` care să suporte un număr rezonabil de constructori și metodele `add()`, `subtract()`, `multiply()`, `divide()`, `equals()`, `compareTo()` și `toString()`. Mențineți `IntType` sub forma unui șir de cifre suficient de mare. Pentru această clasă operația dificilă este împărțirea, urmată îndeaproape de înmulțire.
4. Implementați o clasă simplă numită `Date`. Clasa va trebui să reprezinte orice dată între 1 Ianuarie 1800 și 31 Decembrie 2500; scădeți două date; incrementați o dată cu un număr de zile; comparați două date folosind metodele `equals()` și `compareTo()`. O dată este reprezentată intern ca numărul de zile care au trecut de la o anumită dată, care în cazul nostru este prima zi din 1800. Având în vedere acest lucru, toate metodele, mai puțin constructorul și `toString()` vor fi banale.

Indicație

Regula pentru ani bisecți este: un an este bisect dacă este divizibil prin 4, dar nu prin 100 sau dacă este divizibil cu 400. Astfel 1800, 1900, 2100 nu sunt ani bisecți, dar 2000 este. Constructorul trebuie să verifice validitatea datei, ca și `toString()`. Data ar putea deveni incorectă dacă operatorul de incrementare sau scădere ar face-o să iasă din limitele specificate.

Odată ce am decis asupra specificației, putem trece la implementare.

Partea oarecum dificilă este conversia între reprezentarea internă și cea externă a datei. Urmează acum un posibil algoritm.

Creați două șiruri care să fie atribute statice. Primul șir, pe care îl putem numi `zilePanaLaIntaiALunii` va avea 12 elemente, reprezentând numărul de zile până la prima zi din fiecare lună a unui an nebisect. Astfel, el va conține 0, 31, 59, 90 etc. Cel de-al doilea șir, `zilePanaLa1Ian` va conține numărul de zile până la începutul fiecărui an, începând cu 1800. Astfel, el va conține 0, 365, 730, 1095, 1460, 1826 etc, deoarece 1800 nu este an bisect, dar 1804 este. Programul va trebui să inițializeze acest șir o singură dată folosind un inițializator static. Veți putea apoi folosi acest șir pentru conversia din reprezentarea internă în reprezentarea externă.

5. Moștenire

Știință este orice disciplină în care nebunul generației actuale poate să treacă dincolo de punctul atins de geniul generației anterioare.

Max Gluckman

Așa cum am menționat în capitolul anterior, unul dintre principalele scopuri ale programării orientate pe obiecte este reutilizarea codului. La fel cum inginerii folosesc aceleași componente din nou și din nou în proiectarea circuitelor, programatorii au posibilitatea să refolosească și să extindă obiectele, în loc să le reimplementeze. În limbajele de programare orientate pe obiecte, mecanismul fundamental pentru re folosirea codului este *moștenirea*. Moștenirea ne permite să extindem funcționalitatea unui obiect. Cu alte cuvinte, putem crea noi obiecte cu proprietăți extinse (sau restrânse) ale tipului original, formând astfel o ierarhie de clase. De asemenea, așa cum vom vedea pe parcursul acestui capitol, moștenirea este mecanismul pe care Java îl folosește pentru a implementa metode și clase generice (vezi subcapitolul 5.6).

În acest capitol vom prezenta:

- Principiile generale ale moștenirii, inclusiv polimorfismul;
- Cum este implementată moștenirea în Java;
- Cum poate fi derivată o colecție de clase dintr-o singură clasă abstractă;
- *Interfața*, care este un caz particular de clasă;
- Cum se poate realiza programarea generică în Java folosind interfețe;
- Ce sunt și cum se folosesc clasele interioare;

- Cum se pot identifica tipurile de date în faza de execuție a programului (engl. RunTime Type Identification, RTTI).

5.1 Ce este moștenirea?

Moștenirea este principiul fundamental al programării orientate pe obiecte care permite re folosirea codului între clasele înrudite. Moștenirea modelează relații de tipul ESTE-UN (sau ESTE-O). Într-o relație de tip ESTE-UN, spunem despre clasa derivată că ESTE-O variațiune a clasei de bază. De exemplu, Cerc ESTE-O Curba, iar Masina ESTE-UN Vehicul. În schimb, o Elipsa NU-ESTE-UN Cerc. Relațiile de moștenire formează *ierarhii*. De exemplu, putem extinde clasa Masina la MasinaStraina (pentru care se plătește vamă) și MasinaAutohtona (pentru care nu se plătește vamă) etc.

Un alt tip de relație între obiecte este relația ARE-UN sau ESTE-COMPUS-DIN. De exemplu, Masina ARE-UN Volan. Această relație nu are proprietățile standard dintr-o ierarhie de moștenire, motiv pentru care nu trebuie modelată prin moștenire, ci prin *agregare*, componentele devenind simple câmpuri de tip *private* ale clasei.

Chiar și J2SDK folosește din plin moștenirea pentru a-și implementa propriile biblioteci de clase. Un exemplu relativ familiar îl constituie *excepțiile*, prezentate pe larg în capitolul următor. Java definește clasa `Exception`. Așa cum am văzut deja, există mai multe tipuri de excepții, printre care putem aminti `NullPointerException` și `ArrayIndexOutOfBoundsException`. Fiecare excepție constituie o clasă separată, dar ele au totuși trăsături comune, cum ar fi de exemplu metodele `toString()` sau `printStackTrace()`, utilizate pentru a furniza mesaje de eroare utile în depanare.

Moștenirea modelează în acest caz o relație de tip ESTE-UN. De exemplu, `NullPointerException` ESTE-O `Exception`. Datorită relației de tip ESTE-UN, proprietatea fundamentală a moștenirii garantează că orice metodă aplicabilă lui `Exception` poate fi aplicată și lui `NullPointerException`. Mai mult decât atât, un obiect de tip `NullPointerException` poate să fie referit de către o referință de tip `Exception` (reciproca nu este adevărată!). Prin urmare, deoarece metoda `toString()` este o metodă disponibilă în clasa `Exception`, vom putea întotdeauna scrie:

```
1 public void printException(Exception e)
2 {
3     if (e instanceof NullPointerException)
4     {
5         System.out.println(e.toString());
6     }
```

116


```
7 }
```

Dacă în faza de execuție e referă un obiect `NullPointerException`, atunci `e.toString()` este un apel corect. Funcție de modul de implementare al ierarhiei de clase, metoda `toString()` ar putea fi invariantă sau ar putea fi *specializată* pentru fiecare clasă distinctă. Atunci când o metodă este invariantă în cadrul unei ierarhii, adică are aceeași funcționalitate pentru toate clasele din ierarhie, nu mai este necesar să rescriem implementarea acelei metode pentru fiecare clasă (ea este pur și simplu moștenită).

Apelul lui `toString()` din exemplul de mai sus mai ilustrează un alt principiu important al programării orientate pe obiecte, cunoscut sub numele de *polimorfism*. O variabilă referință care este polimorfică poate să refere obiecte de tipuri diferite. Atunci când se aplică o metodă referinței, se selectează automat operația adecvată pentru tipul obiectului care este referit în acel moment. În Java, toate tipurile referință sunt polimorfice. De exemplu, să presupunem că avem o referință `e` de tip `Exception`, căreia i se aplică `toString()`. În momentul execuției, mașina virtuală Java va vedea care este obiectul efectiv referit de `e` (care poate fi de exemplu `NullPointerException` sau `ArithmeticException`) și va apela metoda `toString()` a acelui obiect. Acest proces este cunoscut sub numele de *legare târzie* sau *legare dinamică* (engl. “late binding”). Mai multe elemente despre polimorfism vom prezenta în paragraful 5.2.3.

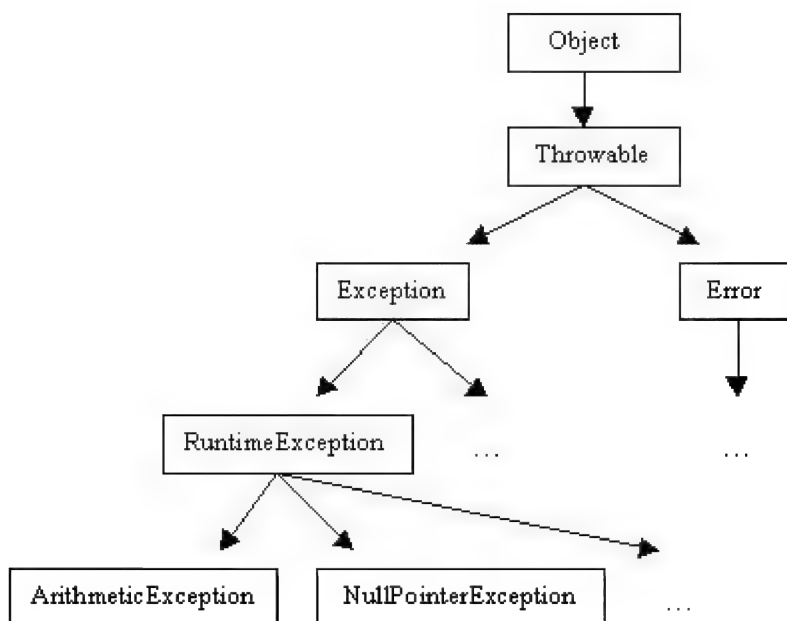
În cazul moștenirii avem o clasă de bază din care sunt derivate alte clase. Clasa de bază reprezintă temelia principiului de moștenire. O *clasă derivată* moștenește toate proprietățile clasei de bază, însemnând că toți membrii publici ai clasei de bază devin membri publici ai clasei derivate (și membrii private sunt moșteniți, dar aceștia nu sunt accesibili în mod direct). Clasa derivată poate să adauge noi atribute și metode și poate modifica semnificația metodelor moștenite. Fiecare clasă derivată este o clasă complet nouă. Clasa de bază nu este în nici un fel afectată de modificările aduse în clasele derivate. Astfel, la crearea clasei derivate este imposibil să se strice ceva în clasa de bază.

O clasă derivată este compatibilă ca tip cu clasa de bază, ceea ce înseamnă că o variabilă referință de tipul clasei de bază poate referi un obiect al clasei derivate, dar nu și invers. Clasele surori (cu alte cuvinte clasele derivate dintr-o clasă comună) nu sunt compatibile ca tip.

Așa cum am menționat anterior, folosirea moștenirii generează de obicei o ierarhie de clase. **Figura 5.1** prezintă o mică parte din ierarhia de excepții a limbajului Java. Remarcați faptul că `NullPointerException` este indirect derivată din `Exception`. Acest lucru nu constituie nici o problemă, deoarece relațiile de tipul ESTE-UN sunt tranzitive. Cu alte cuvinte, dacă `X ESTE-UN`

Y și Y ESTE-UN Z, atunci X ESTE-UN Z. Ierarhia `Exception` ilustrează în același timp designul clasic în care se extrag caracteristicile comune în clasa de bază, urmând ca specializările să fie adăugate în clasele derivate. Într-o astfel de ierarhie, clasa derivată este o *subclasă* a clasei de bază, iar clasa de bază este o *superclasă* a clasei derivate. Aceste relații sunt tranzitive; mai mult, operatorul `instanceof` funcționează pentru subclase. Astfel, dacă `obj` este de tipul X (și nu e `null`), atunci expresia `obj instanceof Z` este adevărată.

Figura 5.1: O parte a ierarhiei de excepții a limbajului Java



În următoarele secțiuni vom examina următoarele probleme:

- Care este sintaxa folosită pentru a deriva o clasă nouă dintr-o clasă de bază?
- Cum afectează acest lucru statutul membrilor private sau public?
- Cum precizăm faptul că o metodă este invariantă pentru o ierarhie de clase?

- Cum factorizăm aspectele comune într-o clasă abstractă, pentru a crea apoi o ierarhie?
- Putem deriva o clasă nouă din mai multe clase (moștenire multiplă)?
- Cum se folosește moștenirea pentru a implementa codul generic?

Câteva dintre aceste subiecte sunt ilustrate prin implementarea unei clase `Shape` din care vom deriva clasele `Circle`, `Square` și `Rectangle`. Ne vom folosi de acest exemplu pentru a vedea modalitatea în care Java implementează polimorfismul, dar și pentru a vedea cum poate fi folosită moștenirea pentru a implementa metode generice.

5.2 Sintaxa de bază Java

O clasă derivată moștenește toate proprietățile clasei de bază. Clasa derivată poate apoi să adauge noi atribute, să redefinească metode sau să adauge noi metode. Fiecare clasă derivată este o clasă complet nouă. Aspectul general al unei clase derivate este prezentat în **Listing 5.1**. Clauza `extends` declară faptul că o clasă este derivată dintr-o (extinde o) altă clasă.

Listing 5.1: Aspectul general al unei clase derivate

```
1 public class Derived extends Base
2 {
3     //membrii (public sau protected) ai clasei Base care
4     //nu sunt listati mai jos vor fi mosteniti nemodificati,
5     //cu exceptia constructorului;
6
7     //membri public ai clasei Derived;
8     //constructori, daca cel implicit nu este suficient;
9     //metode din Base a caror implementare este
10    //modificata;
11    //metode publice aditionale;
12
13    //membri private ai clasei Derived;
14    //atribute si metode private;
15 }
```

Iată o scurtă descriere a unei clase derivate:

- În general, toate atributele sunt `private`, deci atributele suplimentare vor fi adăugate clasei derivate prin precizarea lor în secțiunea `private`;
- Orice metodă non-`private` din clasa de bază care nu este precizată în clasa derivată este moștenită nemodificat, cu excepția constructorilor;

- Orice metodă din clasa de bază care este definită în secțiunea public a clasei derivate este redefinită. Noua metodă va fi aplicată obiectelor din clasa derivată;
- Metodele public din clasa de bază nu pot fi redefinite în secțiunea private a clasei derivate;
- Clasei derivate îi putem adăuga metode suplimentare.

5.2.1 Reguli de vizibilitate

Știm deja că orice membru care este declarat `private` este accesibil doar în cadrul metodelor clasei. Rezultă deci că nici un membru `private` din clasa de bază nu va fi direct accesibil în clasa derivată (el va putea fi accesat prin intermediul modificatorilor și accesoriilor moșteniți de la clasa de bază).

Există situații în care clasa derivată trebuie să aibă acces la membrii clasei de bază. Există două opțiuni pentru a realiza acest lucru. Prima este aceea de a utiliza accesul de tip `public` sau *friendly*. Totuși, accesul de tip `public` permite accesul și altor clase, pe lângă clasele derivate și încalcă principiul ascunderii informației. Pe de altă parte, accesul de tip *friendly* funcționează doar dacă clasa derivată este în același pachet cu clasa de bază, lucru care nu este obligatoriu (vezi capitolul 4.4.4).

Dacă dorim să restrângem accesul unor membri, astfel încât ei să fie vizibili doar pentru clasele derivate, putem să îi declarăm ca fiind de tip `protected`. Un membru de tip `protected` este `private` pentru toate clasele cu excepția claselor derivate și a claselor din același pachet. În consecință vizibilitatea membrilor de tip `protected` este mai lejeră decât a membrilor de tip *package friendly*. Declararea atributelor ca fiind `public` sau `protected` încalcă principiile încapsulării și ascunderii informației, fiind folosită doar din motive de comoditate. De obicei, este preferabil să se scrie modificatori și accesorii (*setter*-i și *getter*-i) sau să se folosească accesul de tip *friendly*. Totuși, dacă o declarație `protected` vă scutește de scrierea de cod stufos, atunci se poate recurge la ea.

5.2.2 Constructor și `super`

Fiecare clasă derivată trebuie să își definească proprii constructori. Dacă nu se scrie nici un constructor, Java va genera un constructor implicit fără parametri. Acest constructor va apela constructorul fără parametri al clasei de bază (dacă nu există, se obține eroare la compilare!) pentru membrii care au fost

moșteniți, după care va aplica inițializarea implicită pentru atributele adăugate (adică 0/false pentru tipurile primitive și null pentru tipurile referință).

Construirea unui obiect al unei clase derivate are loc prin construirea prealabilă a porțiunii moștenite (constructorul clasei derivate apelează automat constructorul implicit al clasei de bază). Acest lucru este natural, deoarece principiul încapsulării afirmă că partea moștenită este o entitate unică, iar constructorul clasei de bază ne spune cum să inițializăm această entitate.

Constructorii clasei de bază pot fi apelați explicit prin metoda `super()`. Astfel, constructorul implicit pentru o clasă derivată are de fapt forma:

```
1 public Derived()
2 {
3     super();
4 }
```

Metoda `super` poate fi utilizată și pentru a apela constructori cu parametri. De exemplu, dacă avem o clasă de bază care are un constructor cu doi parametri de tip `int`, atunci constructorul clasei derivate va avea în general forma:

```
1 public Derived(int x, int y)
2 {
3     super(x, y);
4     // alte instructiuni
5 }
```

Metoda `super()` poate să apară doar în *prima* linie dintr-un constructor. Dacă nu se face un apel explicit al lui `super()`, compilatorul va realiza automat un apel al metodei `super()` fără parametri chiar la începutul constructorului. Este important de remarcat că apelul implicit este către constructorul *implicit* (fără parametri) al clasei de bază. Dacă acest constructor nu există (să nu uităm că el se generează automat doar dacă **nu** există alți constructori), se obține eroare la compilare. De exemplu, clasa `Base` de mai jos definește un constructor cu un parametru de tip `int`:

```
1 public class Base
2 {
3     public Base(int x)
4     {
5         System.out.println("Base(int) called.");
6     }
7 }
```

Clasa `Derived`, extinde clasa `Base`, redefinește constructorul cu parametru de tip `int` și definește o metodă `main()` care creează un obiect de tip `Derived`:

```
1 public class Derived extends Base
2 {
```

```
3 public Derived ( int x )
4 {
5     System.out.println("Derived(int) called." );
6 }
7
8 public static void main( String [] args )
9 {
10     new Derived (2) ;
11 }
12 }
```

Aparent, nimic nu este în neregulă cu aceste clase. Totuși, construirea obiectului `Derived`, presupune construirea în prealabil a obiectului `Base`, pe care acesta îl extinde. Cum constructorul din `Derived` nu conține un apel explicit al constructorului din `Base`, compilatorul va adăuga automat un apel către constructorul implicit. Dar, surpriză! Acest constructor nu există și nici nu a fost generat implicit, deci se va obține o eroare la compilare, în care suntem anunțați cu părere de rău că “nu a fost găsit constructorul `Base()`”. Corect este ca prima linie din constructorul clasei `Derived` să fie un apel către constructorul cu parametru de tip `int` din clasa `Base`:

```
1 public Derived ( int x )
2 {
3     super(x) ;
4     System.out.println("Derived(int) called." );
5 }
```

La execuția clasei `Derived` se va afișa în consolă:

```
Base(int) called.
Derived(int) called.
```

5.2.3 Redefinirea metodelor. Polimorfism

Polimorfismul, al treilea concept fundamental al programării orientate pe obiecte, alături de încapsulare și moștenire, crează adeseori un anumit grad de confuzie în rândul programatorilor începători. Vom urmări în acest paragraf să clarificăm această importantă noțiune, lucru absolut necesar pentru o înțelegere profundă a programării orientate pe obiecte.

Capacitatea de a asuma diferite forme poartă numele de “polimorfism”. Apa oferă un bun exemplu de polimorfism în lumea reală: apare sub formă solidă (gheața), lichidă, sau gazoasă (vapori de apă). În Java, polimorfismul înseamnă că o singură variabilă referință poate fi folosită pentru a desemna mai multe obiecte - instanțe ale unor clase similare¹ - în diferite momente ale execuției

¹a se citi clase derivate direct sau indirect din aceeași clasă de bază.

unui program. Când o referință *x* este folosită pentru a invoca o metodă a unui obiect (de exemplu, *x.metodaMea()*), metoda precisă care va fi apelată depinde de obiectul pe care variabila referință îl indică în acel moment.

O variabilă referință poate referi un obiect a cărui clasă este similară cu tipul variabilei referință. De exemplu, să presupunem că avem clasele *Animal*, *Pisica*, *Soarece*, *Caine*, definite ca mai jos:

Listing 5.2: Exemplu de polimorfism

```

1 public class Animal
2 {
3     public String caracteristica()
4     {
5         return "necunoscuta";
6     }
7 }
8
9 public class Soarece extends Animal
10 {
11     public String caracteristica()
12     {
13         return "are culoarea gri";
14     }
15 }
16
17 public class Pisica extends Animal
18 {
19     public String caracteristica()
20     {
21         return "miauna";
22     }
23 }
24
25 public class Caine extends Animal
26 {
27     public String caracteristica()
28     {
29         return "latra";
30     }
31 }
32
33 public class PolimorfismEx
34 {
35     public static void main(String args[])
36     {
37         Animal a = new Soarece();
38         System.out.println("Caracteristica primului animal: " +
39                             a.caracteristica());
40     }

```

```
41         //alte instructiuni...
42
43         a = new Caine();
44         System.out.println("Caracteristica celui de-al doilea" +
45                             " animal: " + a.caracteristica());
46
47         //alte instructiuni...
48     }
49 }
```

Ceea ce atrage atenția asupra acestui program este faptul că toate cele patru metode `caracteristica()` au aceeași semnătură (nume și listă de parametri). Acest lucru este permis deoarece nu există două metode cu aceeași semnătură în *aceeași* clasă. Cum trei dintre aceste clase (*Soarece*, *Caine*, *Pisica*) extind o a patra clasă (*Animal*), spunem că metodele `caracteristica()` din clasele derivate *redefinesc* (engl. *override*) metoda `caracteristica()` din clasa de bază.

Să analizăm acum instrucțiunea următoare:

```
Animal a = new Soarece();
```

În această situație, variabila referință *a* care este de tipul *Animal* desemnează un obiect de tipul *Soarece*, care ESTE-UN *Animal* prin moștenire (prin urmare, atribuirea este corectă). Mai departe, întâlnim în cadrul aceleiași secvențe de cod:

```
...
```

```
a = new Caine();
```

Cu alte cuvinte, variabila *a* referă (la momente diferite de timp) obiecte de tipuri diferite, dar similare (prin faptul că sunt derivate din aceeași clasă: *Animal*). Pentru a vă convinge de acest lucru, rulați acest program. El va afișa următoarele date:

```
Caracteristica primului animal: are culoarea gri
Caracteristica celui de-al doilea animal: latra
```

După cum se poate observa, apelurile metodei `caracteristica()`, prin intermediul referinței *a*, au determinat apelarea metodelor corespunzătoare din clasele *Soarece* și *Caine* (și prin urmare, două afișări diferite), deoarece la momentul fiecărui apel, variabila *a* referea un obiect de tip diferit. Poate că unii dintre dumneavoastră se așteptau ca singura metodă apelată să fie cea din clasa *Animal*, motiv pentru care rezultatul vi se pare anormal, dar el va fi perfect explicabil după ce vom detalia noțiunea de apel polimorfic al unei metode.

Ca o paranteză, trebuie remarcat faptul că inversarea rolurilor celor două clase nu este posibilă în cadrul acestui exemplu (normal de altfel, deoarece un animal nu este în mod obligatoriu un șoarece):

```
Soarece s = new Animal(); //INCORECT...
```

După cum am văzut anterior, în cadrul subcapitolului 5.1, moștenirea permite tratarea unui obiect ca fiind de tipul propriu *sau* de tipul de bază (din care este derivat tipul propriu). Această caracteristică este esențială deoarece permite mai multor tipuri (derivate din același tip de date) să fie tratate unitar, ca și cum ar fi un singur tip, motiv pentru care aceeași secvență de cod va fi funcțională pentru toate aceste tipuri. Pe de altă parte, apelul polimorfic al unei metode permite unui tip să exprime distincția față de un alt tip similar, atâta timp cât amândouă sunt derivate din aceeași clasă de bază. Distincția este exprimată prin diferența în comportamentul metodelor care pot fi apelate prin intermediul clasei de bază (în exemplul nostru, este vorba despre metoda `caracteristica()`).

Polimorfismul permite scrierea de cod doar în raport cu clasa de bază, ca și cum “am fi uitat” de clasele derivate. Totuși, cei mai mulți programatori, în special cei cu experiență în programarea procedurală, întâmpină anumite dificultăți în înțelegerea modului de funcționare a polimorfismului.

Dificultatea poate fi întâlnită și în cadrul programului din exemplul anterior. Afișarea celor două mesaje este evident cea dorită, deși parcurgând codul sursă al programului, am putea avea impresia că programul nu va funcționa în acest mod. Dacă privim mai atent linia 38, observăm că metoda `caracteristica()` este apelată prin intermediul unei referințe de tipul `Animal`. Așa că ne punem în mod logic întrebarea: este posibil să știe compilatorul că variabila referință de tipul `Animal` referă un obiect de tipul `Soarece`, din moment ce se apelează metoda acesteia din urmă? De unde știe compilatorul că este referit un obiect de tip `Soarece`, și nu un obiect de alt tip (de exemplu, `Pisica` sau `Caine`)? Răspunsul este că, de fapt, compilatorul nu știe acest lucru. Pentru a înțelege mai bine această problemă, este necesară o examinare mai atentă a noțiunii de *legare*.

Conectarea unui apel de metodă de un anumit corp de metodă poartă numele de *legare* (engl. binding). Când legarea are loc înainte de rularea programului respectiv (cu alte cuvinte, în faza de compilare), spunem că este vorba despre o *legare statică* (engl. early binding). Termenul este specific programării orientate pe obiecte. În programarea procedurală (gen programarea în C sau Pascal), noțiunea de legare statică nici nu există, pentru că toate legăturile se fac în mod static.

În programul anterior, compilatorul nu știe ce metodă `caracteristica()`

(din clasa `Animal`, `Soarece`, `Pisica` sau `Caine`?) să apeleze, având doar o referință de tipul `Animal`. Soluția constă însă în *legarea târzie* (engl. late binding). Cunoscută și sub numele de *legare dinamică* sau *legare în faza de execuție* (engl. dynamic binding, run-time binding), legarea târzie permite determinarea în faza de execuție a tipului obiectului referit de o variabilă referință și apelarea metodei specifice acestui tip (în exemplul nostru, metoda `caracteristica()` din clasa `Soarece`). Mecanismul de apelare a metodelor determină în mod corect metoda care trebuie apelată și o apelează. Legarea este un fenomen care are loc automat - nu este atribuția programatorului să decidă dacă o metodă este apelată prin legare statică sau dinamică.

Am ajuns la momentul în care trebuie să conturăm diferența dintre *redefinirea* (suprascierea) unei metode și *supraîncărcarea* ei. Cu alte cuvinte, *overriding* vs. *overloading*. Finalul capitolul 2, secțiunea 2.6, a prezentat noțiunea de supraîncărcare a metodelor. Recapitulând pe scurt informațiile precizate acolo, supraîncărcarea metodelor permite existența în interiorul aceleași clase a mai multor metode cu același nume, dar cu lista *diferită* de parametri (prin urmare, cu semnături diferite). Astfel, putem avea o metodă `int max(int a, int b)` și o metodă `int max(int a, int b, int c)`, ambele în cadrul aceleiași clase. Evident că, analog, putem avea prima metodă în cadrul unei clase de bază și cea de-a doua în cadrul unei clase derivate din clasa de bază. Pe de altă parte, noțiunea de suprasciere (redefinire) se bazează tocmai pe ideea că metodele trebuie să aibe *aceeași* semnătură și să se afle în clase derivate din clasa de bază (vezi exemplul din această secțiune). Diferența dintre cele două concepte este în acest moment clar definită.

Cele două concepte pot provoca uneori greșeli de programare greu detectabile, în cazul în care nu sunt folosite corect. Iată un exemplu sugestiv: să presupunem că metoda `caracteristica()` din clasa `Soarece` ar fi avut de fapt următoarea definiție:

```
1 public String caracteristica(int i)
2 {
3     return "are blana gri";
4 }
```

Poate că acest antet ar fi fost o greșeală neintenționată a programatorului, sau pur și simplu, ar fi fost ceva intenționat. Ce s-ar fi întâmplat în această situație la execuția liniei 38 din exemplul nostru? Cert este că aceasta nu ar fi fost interpretată drept o eroare de către compilator. Programul ar fi rulat și ar fi afișat rezultatele:

```
Caracteristica primului animal: necunoscuta
```

```
126
```

Caracteristica celui de-al doilea animal: latra

Poate că vi se pare ciudat rezultatul execuției versiunii modificate a programului. Să vedem ce s-a întâmplat de fapt. Deoarece în clasa *Soarece*, noua metodă *caracteristica()* are o semnătură diferită față de metoda cu același nume a clasei de bază, am realizat o *supraîncărcare* a acelei metode, și nu o *redefinire* a ei. Astfel clasa *Soarece* are acum două metode *caracteristica()* diferite: una moștenită prin derivarea din clasa de bază și cealaltă prin supraîncărcarea metodei din clasa de bază. În urma apelului din linia 38 se va executa însă metoda moștenită din clasa de bază, și nu cea *supraîncărcată*, deoarece aceea corespunde ca semnătură. Astfel, mesajul afișat este “necunoscuta”.

Deși se intenționase o redefinire a acelei metode, datorită faptului că s-a greșit la definirea antetului ei, rezultatul nu a fost cel așteptat. Totuși, compilatorul a presupus că intenția a fost de a supraîncărca metoda și nu de a o redefini, motiv pentru care nu a afișat nici o eroare. Evident că acesta nu este un apel polimorfic de metodă. Odată ce a fost depistată eroarea, ea poate fi înlăturată simplu, însă este infinit mai greu în cazul unor aplicații mai complexe. La fel de adevărat este faptul că uneori *exact* acest comportament este dorit. Programatorii avansați vor ști cu siguranță cum să “jongleze” cu cele două noțiuni. Cei începători vor trebui să fie foarte atenți în astfel de situații, pentru a implementa exact comportamentul pe care îl doresc.

Datorită polimorfismului, exemplul nostru poate fi extins, adăugând noi funcționalități prin intermediul unor tipuri noi de date (clase) care moștenesc comportamentul clasei de bază. Altfel spus, putem adăuga clase noi ce extind comportamentul clasei *Animal*, fără ca aceasta să prejudicieze codul existent.

Referitor la modul de utilizare a polimorfismului există o singură problemă ce trebuie tratată cu mare atenție pentru că este generatoare de mari probleme, greu detectabile mai ales în cadrul unor programe de dimensiuni mari: este vorba despre utilizarea metodelor polimorfe în cadrul constructorilor. Deoarece obiectele nu sunt complet construite în această fază a execuției unui program, ceea ce presupune că identificarea tipului obiectelor respective este problematică, apelul polimorfic al unor metode poate crea un comportament imprevizibil și, în cele mai multe situații, nedorit.

Polimorfismul presupune existența unei “fețe” (funcționalitatea comună din clasa de bază) și a mai multor “forme” care folosesc această față (diferitele funcționalități ale metodelor legate dinamic). Rețineți că polimorfismul apare doar atunci când este vorba despre legarea târzie a metodelor. Polimorfismul oferă o nouă viziune asupra modului de decuplare dintre ceea ce face o clasă și cum îndeplinește ea acest lucru. Polimorfismul realizează o decuplare la nivel

de *tipuri*. De asemenea, permite o bună organizare a codului, dar și crearea de programe extensibile în adevăratul sens al cuvântului, care pot fi îmbunătățite ca funcționalitate nu numai în faza de creare a aplicației, ci și pe parcursul existenței ei, când noi facilități sunt de dorit.

5.2.4 Redefinirea parțială a metodelor

Am văzut deja că metodele din clasa de bază pot fi redefinite în clasele derivate prin furnizarea unei metode din clasa derivată care să aibă aceeași semnătură. De asemenea, metoda din clasa derivată nu are dreptul să adauge excepții la lista `throws` (detalii despre `throws` în capitolul următor).

Adeseori, metoda din clasa derivată nu redefineste complet metoda din clasa de bază, ci extinde operațiile pe care aceasta le realizează. Acest proces este numit *redefinire parțială*. Cu alte cuvinte vrem să facem ceea ce face și metoda din clasa de bază, plus încă ceva. Apeluri ale metodei din clasa de bază pot fi realizate folosind din nou `super`. Iată un exemplu:

Listing 5.3: Exemplu de redefinire parțială

```
1 public class Student extends Human
2 {
3     public doWork ()
4     {
5         takeBreak ();           // pauzele lungi si dese
6         super.doWork ();        // cheia marilor
7         takeBreak ();           // succese
8     }
9 }
```

Astfel, metoda `doWork ()` din clasa `Student` extinde metoda `doWork ()` din clasa `Human`, adăugându-i două elemente importante: pauza și pauza... Apelul polimorfic se aplică în mod identic și în cazul redefinirii parțiale.

5.2.5 Metode și clase `final`

Așa cum am precizat anterior, clasa derivată poate să redefinească sau să accepte nemodificate metodele din clasa de bază. În multe cazuri este clar faptul că o anumită metodă trebuie să fie invariantă de-a lungul ierarhiei de clase, ceea ce înseamnă că nici o clasă derivată nu trebuie să o redefinească. În acest caz, putem declara metoda ca fiind de tip `final`, iar ea nu va putea fi redefinită.

Pe lângă faptul că declararea unei metode `final` este o practică bună de programare, ea poate genera și cod mai rapid. Declararea unei metode `final` (atunci când este cazul) este utilă deoarece intențiile noastre devin astfel clare

pentru cititorul programului și al documentației și, pe de altă parte, putem preveni redefinirea accidentală pentru o metodă care nu trebuie să fie redefinită.

Pentru a vedea de ce folosirea lui `final` poate conduce la cod mai eficient, să presupunem că avem o clasă de bază numită `Base` care definește o metodă finală `f()`, iar `Derived` este o clasă care extinde `Base`. Să considerăm metoda:

```
1 public void xxx(Base obj)
2 {
3     obj.f();
4 }
```

Deoarece `f()` este o metodă `final`, nu are nici o importanță dacă în momentul execuției, `obj` referă un obiect de tip `Base` sau un obiect de tip `Derived`; definirea lui `f` este invariantă, deci știm de la început ceea ce `f` va face. O consecință a acestui fapt este că decizia pentru codul care va fi executat se ia încă de la *compilare*, și nu la *execuție*. Este vorba, în mod evident, de o *legare statică*. Deoarece legarea se face la compilare și nu la execuție, programul ar trebui să ruleze mai repede. Dacă acest fapt este sau nu perceptibil efectiv (ținând cont de viteza de prelucrare a procesoarelor din generația actuală) depinde de numărul de ori în care evităm deciziile din timpul execuției programului. În Java, orice legare de metode este polimorfică, prin intermediul legării târzii, cu excepția situației în care metoda este declarată `final`.

Un corolar al acestei observații îl constituie faptul că dacă `f` este o metodă `final` banală, cum ar fi un accesoriu pentru un atribut, compilatorul ar putea să înlocuiască apelul lui `f()`, direct cu corpul funcției. Există foarte multe optimizări pe care le face compilatorul Java pentru a mări eficiența codului, iar aceasta este doar una dintre ele. Astfel, apelul funcției va fi înlocuit cu o singură linie care accesează un atribut, economisindu-se astfel timp (apelul unei metode și transmiterea de parametri sunt operații destul de costisitoare). Dacă `f()` nu ar fi fost declarată `final`, acest lucru ar fi fost imposibil, deoarece `obj` ar fi putut referi un obiect al unei clase derivate, pentru care definirea lui `f()` ar fi putut fi diferită.

Este interesant de remarcat faptul că, deoarece metodele statice nu au un obiect care să le controleze, apelul lor este rezolvat încă de la compilare folosind legarea statică.

Similare metodelor `final` sunt clasele `final`. O clasă `final` nu mai poate fi extinsă. În consecință, toate metodele unei astfel de clase sunt automat metode `final`. Ca un exemplu, clasa `Integer` din pachetul `java.lang` este o clasă `final`. De remarcat faptul că dacă o clasă are doar membri `final`, ea nu este neapărat o clasă `final`.

5.2.6 Metode și clase abstracte

Până acum am văzut faptul că unele metode sunt invariante de-a lungul ierarhiei de clase (metodele `final`), iar alte metode își modifică semnificația de-a lungul ierarhiei. O a treia posibilitate este ca o metodă din clasa de bază să aibă sens *doar* pentru clasele derivate, și vrem ca ea să fie obligatoriu definită în clasele derivate. Totuși, implementarea metodei nu are nici un sens pentru clasa de bază. În această situație, putem declara metoda ca fiind abstractă.

O *metodă abstractă* este o metodă care declară funcționalități care vor trebui neapărat implementate până la urmă în clasele derivate. Cu alte cuvinte o metodă abstractă spune ceea ce obiectele derivate trebuie să facă. Totuși, ea nu furnizează nici un fel de implementare, ci fiecare clasă derivată trebuie să vină cu propria implementare.

O clasă care are cel puțin o metodă abstractă este o *clasă abstractă*. Java pretinde ca toate clasele abstracte să fie definite explicit ca fiind abstracte. Atunci când o clasă derivată nu redefineste o metodă abstractă, metoda va fi moștenită abstractă și în clasa derivată. În consecință dacă o clasă care nu intenționăm să fie abstractă, nu redefineste toate metodele abstracte, compilatorul va detecta inconsistența și va genera un mesaj de eroare.

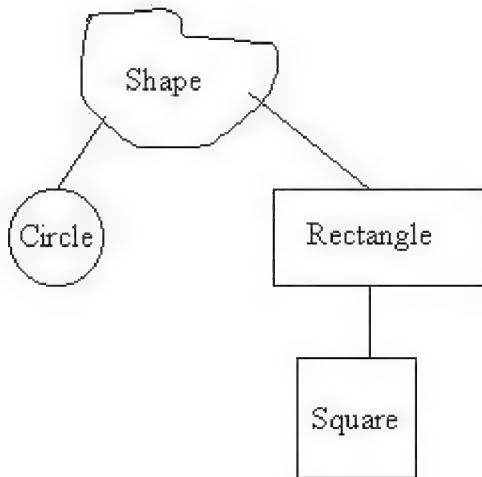
Un exemplu simplu de clasă abstractă este clasa `Shape` (rom. formă sau curbă), pe care o vom folosi ca exemplu în cadrul acestui capitol. Din `Shape` vom deriva forme specifice cum ar fi `Circle` sau `Rectangle`. Putem deriva apoi `Square` ca un caz particular de `Rectangle`. **Figura 5.2** prezintă ierarhia de clase care rezultă.

Clasa `Shape` poate să aibă membri care să fie comuni pentru toate clasele. Într-un exemplu mai extins, aceasta ar putea include coordonatele extremităților obiectului. Clasa ar putea declara și defini metode cum ar fi `positionOf`, pentru a determina poziția formei. Aceste metode sunt independente de tipul formei, motiv pentru care `positionOf` ar fi o metodă `final`. Clasa definește metode care se aplică fiecărui obiect în parte. Unele dintre aceste metode nu au nici un sens pentru clasa abstractă `Shape`. De exemplu, este dificil de calculat aria unui obiect oarecare; în consecință metoda `area` va fi declarată abstract.

Așa cum am menționat anterior, existența a cel puțin o metodă abstractă, face clasa să devină și ea abstractă, deci ea nu va putea fi instanțiată. Astfel, nu vom putea crea un obiect de tip `Shape` și vom putea crea doar obiecte derivate. Totuși, ca de obicei, o referință de tip `Shape` poate să refere orice formă concretă derivată, cum ar fi `Circle` sau `Rectangle`. Exemplu:

```
Shape a,b;
a = new Circle(3.0);           // corect
```

Figura 5.2: Ierarhia de clase pentru forme



```
b = new Shape("circle");    //INCORECT!
```

Codul din **Listing 5.4** prezintă clasa abstractă `Shape`. La linia 13, declarăm o variabilă de tip `String` care reține tipul formei, folosită doar în clasele derivate. Atributul este de tip `private`, deci clasele derivate nu au acces direct la el. Restul clasei cuprinde o listă de metode.

Constructorul nu va fi apelat niciodată direct, deoarece `Shape` este o clasă abstractă. Avem totuși nevoie de un constructor care să fie apelat din clasele derivate pentru a inițializa atributele `private`. Constructorul clasei `Shape` stabilește valoarea atributului `name`.

Linia 15 declară metoda abstractă `area`. `area` este o metodă abstractă, deoarece nu putem furniza nici un calcul implicit al ariei pentru o clasă derivată care nu își definește propria metodă de calcul a ariei.

Metoda de comparație din liniile 22-25 nu este abstractă, deoarece ea poate fi aplicată în același mod pentru toate clasele derivate. De fapt, definirea ei este invariantă de-a lungul ierarhiei, de aceea am declarat-o `final`. Parametrul `rhs` (de la "right-hand-side") reprezintă un alt obiect de tip `Shape`, a cărui arie se compară cu cea a obiectului curent. Este interesant de remarcat faptul

că variabila `rhs` poate să refere orice instanță a unei clase derivate din `Shape` (de exemplu o referință a clasei `Rectangle`). Astfel este posibil ca folosind această metodă să comparăm aria obiectului curent (care poate fi, de exemplu, o instanță a clasei `Circle`) cu aria unui obiect de alt tip, derivat din `Shape`. Acesta este un exemplu excelent de folosire a polimorfismului.

Metoda `toString` din liniile 27-30 afișează numele formei și aria ei. Ca și metoda de comparație `lessThan`, ea este invariantă de-a lungul ierarhiei (nu face decât să concateneze două stringuri, indiferent de natura formei), de aceea a fost declarată *final*.

Listing 5.4: Clasa de bază abstractă `Shape`

```

1  /* Clasa de baza abstracta pentru forme
2  *
3  * CONSTRUIREA: nu este permisa, Shape fiind abstracta.
4  * Constructorul cu un parametru este furnizat ptr. clasele
5  * derivate.
6  *-----metode publice-----
7  * double area()          --> Intoarce aria (abstracta)
8  * boolean lessThan      --> Compara doua forme dupa arie
9  * String toString       --> Metoda uzuala pentru scriere
10 *
11 abstract class Shape
12 {
13     private String name;
14
15     abstract public double area();
16
17     public Shape( String shapeName)
18     {
19         name = shapeName;
20     }
21
22     final public boolean lessThan(Shape rhs)
23     {
24         return area() < rhs.area();
25     }
26
27     final public String toString()
28     {
29         return name + ", avand aria " + area();
30     }
31 }
```

Înainte de a trece mai departe, să rezumăm cele patru tipuri de metode ale unei clase:

1. *Metode finale*. Apelul lor este rezolvat încă de la compilare. Folosim

metode `final` doar atunci când metoda este invariantă de-a lungul ierarhiei (adică atunci când metoda nu este niciodată redefinită);

2. *Metode abstracte.* Apelul lor este rezolvat în timpul execuției. Clasa de bază nu furnizează nici o implementare a lor și este abstractă. Clasele derivate trebuie fie să implementeze metoda, fie devin ele însele abstracte;
3. *Metode statice.* Apelul este rezolvat la compilare, deoarece nu există obiect care să le controleze;
4. *Alte metode.* Apelul este rezolvat la execuție. Clasa de bază furnizează o implementare implicită care fie va fi redefinită (parțial sau total) în clasele derivate, fie acceptată nemodificată.

5.3 Exemplu: Extinderea clasei Shape

În această secțiune vom implementa clasele derivate din clasa `Shape` și vom prezenta cum sunt ele utilizate într-o manieră polimorfică. Iată enunțul problemei:

Sortare de forme. *Se citesc N forme (cercuri, dreptunghiuri sau pătrate). Să se afișeze formele ordonate după arie.*

Implementarea claselor `Circle`, `Rectangle` și `Square`, prezentată în **Listing 5.5** este simplă și nu ilustrează aproape nici un concept pe care să nu-l fi prezentat deja. Singura noutate este că clasa `Square` este derivată din clasa `Rectangle` care este, la rândul ei, derivată din `Shape`. La implementarea fiecărei dintre aceste clase trebuie:

1. să definim un nou constructor;
2. să examinăm fiecare metodă care nu este `final` sau `abstract` pentru a vedea dacă dorim să o acceptăm nemodificată. Pentru fiecare astfel de metodă care nu corespunde cu necesitățile clasei trebuie să furnizăm o nouă definire;
3. să definim fiecare metodă abstractă;
4. să adăugăm alte metode dacă este necesar.

Listing 5.5: Codul complet pentru clasele Circle, Rectangle și Square, care va fi salvat în 3 fișiere sursă separate

```

1  /* clasele Circle, Square si Rectangle;
2  * toate sunt derivate din Shape
3  *
4  * CONSTRUCTORI: (a) cu raza ptr. cerc, (b) cu latura
5  * ptr. patrat, (c) cu lungime si latime ptr. dreptunghi
6  *-----metode publice-----
7  * double area()-->Implementeaza metoda abstracta din Shape
8  *
9  * ATENTIE: Fisierul trebuie separat in 3 pentru compilare!
10 */
11 public class Circle extends Shape
12 {
13     private double radius;
14
15     public Circle(double rad)
16     {
17         super("Circle");
18         radius = rad;
19     }
20
21     public double area()
22     {
23         return Math.PI * radius * radius;
24     }
25 }
26
27 public class Rectangle extends Shape
28 {
29     private double length;
30     private double width;
31
32     public Rectangle(double len, double wid)
33     {
34         this(len, wid, "Rectangle");
35     }
36
37     Rectangle(double len, double wid, String name)
38     {
39         super(name);
40         length = len;
41         width = wid;
42     }
43
44     public double area()
45     {
46         return length * width;
47     }

```

```

48 }
49
50 public class Square extends Rectangle
51 {
52     public Square(double side)
53     {
54         super(side, side, "Square");
55     }
56 }

```

Pentru fiecare clasă am scris un constructor **public** simplu care permite inițierea cu dimensiunile de bază (rază pentru cercuri, lungimea laturilor pentru dreptunghiuri și pătrate). Pentru clasa `Rectangle` a fost necesar și un constructor `package-friendly`, care să permită crearea unui obiect `Square`, cu numele corespunzător. Drept urmare, constructorul public al clasei `Rectangle` este și el adaptat, apelând de fapt constructorul `package-friendly` din aceeași clasă.

În constructorul public, vom inițializa mai întâi partea moștenită prin apelul lui `super`. În cazul clasei `Rectangle` acest apel nu este direct, ci prin intermediul constructorului `package-friendly`. Fiecare clasă trebuie să definească o metodă `area()`, deoarece `Shape` a declarat-o ca fiind abstractă. Dacă uităm să scriem o metodă `area` pentru una dintre clase, eroarea va fi detectată încă de la compilare, deoarece - dacă metoda `area()` lipsește - atunci clasa derivată este și ea abstractă. Observați că `Square` este dispusă să accepte metoda `area()` moștenită de la `Rectangle`, de aceea nu o mai redefinește. Logica acestei abordări derivă din faptul că aria unui pătrat se calculează exact ca și aria unui dreptunghi, prin produsul lungimii a două laturi alăturate.

După ce am implementat clasele, suntem gata să rezolvăm problema ordonării, iar pentru aceasta vom folosi un șir de clase `Shape`. Rețineți faptul că prin aceasta nu alocăm memorie pentru nici un obiect de tip `Shape` (ceea ce ar fi ilegal deoarece `Shape` este clasă abstractă), ci se alocă memorie doar pentru un șir de referințe către `Shape`. Aceste referințe vor putea referi obiecte de tip `Circle`, `Rectangle` sau `Square`².

În **Listing 5.6** realizăm exact acest lucru. Mai întâi citim obiectele. La linia 21, apelul lui `readShape()` constă în citirea unui caracter, urmată de dimensiunile figurii și de crearea unui nou obiect de tip `Shape`. **Listing 5.7** prezintă o implementare primitivă a acestei rutine. Observați că în cazul unei erori la citire se creează un cerc de rază 0 și se întoarce o referință la el. O soluție mai elegantă în această situație ar fi fost să definim și să aruncăm o excepție.

²Nici nu se poate inventa o ilustrare mai bună pentru polimorfism (poli = mai multe, morphos = forme). Într-adevăr referința de tip `Shape` (shape = formă) poate referi clase de mai multe (poli) forme (morphos): `Circle`, `Rectangle` și `Square`.

După aceasta, fiecare obiect creat de către `readShape()` este referit de către un element al șirului `shapes`. Se apelează `insertionSort()` (**Listing 5.8**) pentru a sorta formele. În final afișăm șirul rezultat de forme, apelând astfel implicit metoda `toString()`.

Listing 5.6: Rutina `main()` pentru citirea de figuri și afișarea lor în ordine crescătoare

```

1 import java.io.* ;
2
3 class TestShape
4 {
5     private static BufferedReader in;
6
7     public static void main(String[] args)
8     {
9         try
10        {
11            //Citeste numarul de figuri
12            in = new BufferedReader(new
13                InputStreamReader(System.in));
14            System.out.print("Numarul de figuri: ");
15            int numShapes = Integer.parseInt(
16                in.readLine());
17            //citeste formele
18            Shape[] shapes = new Shape[numShapes];
19            for (int i = 0; i < numShapes; ++i)
20            {
21                shapes[i] = readShape();
22            }
23
24            //sortare si afisare
25            insertionSort(shapes);
26            System.out.println("Sortarea dupa arie: ");
27            for (int i = 0; i < numShapes; ++i)
28            {
29                System.out.println(shapes[i]);
30            }
31        }
32        catch(Exception e)
33        {
34            System.out.println(e);
35        }
36    }
37
38    private static Shape readShape()
39    {
40        /* Implementarea in Listing 5.7 */
41    }
42

```

```

43 //sortare prin insertie
44 private static void insertionSort(Shape[] a)
45 {
46     /*Implementarea in Listing 5.8*/
47 }
48 }

```

Listing 5.7: Rutină simplă pentru citirea și returnarea unei noi forme

```

1 //creaza un obiect adecvat de tip Shape functie de
2 //datele de intrare.
3 //utilizatorul introduce 'c', 's' sau 'r' pentru a indica
4 //forma, apoi introduce dimensiunile
5 //in caz de eroare se intoarce un cerc de raza 0
6
7 private static Shape readShape()
8 {
9     double rad;
10    double len;
11    double wid;
12    String s;
13
14    try
15    {
16        System.out.println("Introduceti tipul formei: ");
17        do
18        {
19            s = in.readLine();
20        } while (s.length() == 0);
21
22        switch (s.charAt(0))
23        {
24            case 'c':
25                System.out.println("Raza cercului: ");
26                rad = Integer.parseInt(in.readLine());
27                return new Circle(rad);
28            case 's':
29                System.out.println("Latura patratului: ");
30                len = Integer.parseInt(in.readLine());
31                return new Square(len);
32            case 'r':
33                System.out.println("Lungimea si latimea "
34                    + "dreptunghiului pe linii separate: ");
35                len = Integer.parseInt(in.readLine());
36                wid = Integer.parseInt(in.readLine());
37                return new Rectangle(len, wid);
38            default:
39                System.err.println("Tastati c, r sau s: ");
40                return new Circle(0);
41        }
42    }
43 }

```

```

42     }
43     catch (IOException e)
44     {
45         System.err.println(e);
46         return new Circle(0);
47     }
48 }

```

Listing 5.8: Sortarea prin inserție

```

1 //sortare prin insertie
2 private static void insertionSort(Shape[] a)
3 {
4     for (int p = 1; p < a.length; ++p)
5     {
6         Shape tmp = a[p];
7         int j = p;
8         for (; j > 0 && tmp.lessThan(a[j-1]); --j)
9         {
10             a[j] = a[j-1];
11         }
12         a[j] = tmp;
13     }
14 }

```

5.4 Moștenire multiplă

Toate exemplele prezentate până acum derivau o clasă dintr-o singură clasă de bază. În cazul *moștenirii multiple* o clasă este derivată din mai mult de o clasă de bază. De exemplu, putem avea clasele `Student` și `Angajat`. Din aceste clase ar putea fi derivată o clasă `AngajatStudent`.

Deși moștenirea multiplă pare destul de atrăgătoare, iar unele limbaje (cum ar fi C++) chiar o implementează, ea este îmbibată de subtilități care fac proiectarea claselor deosebit de dificilă. De exemplu, cele două clase de bază ar putea conține metode care au aceeași semnătură, dar implementări diferite sau ar putea avea atribute cu același nume. Care dintre ele ar trebui folosit?

Din aceste motive, Java nu permite moștenirea multiplă. Java furnizează însă o alternativă pentru moștenirea multiplă prin intermediul conceptului de *interfață*.

5.5 Interfețe

Interfața în Java este cea mai abstractă clasă posibilă. Ea constă doar din metode publice abstracte și din attribute statice și finale.

Spunem că o clasă *implementează* o anumită interfață dacă furnizează definiții pentru toate metodele abstracte din cadrul interfeței. O clasă care implementează o interfață se comportă ca și când ar fi extins o clasă abstractă precizată de către acea interfață.

În principiu, diferența esențială dintre o clasă abstractă și o interfață este că, deși amândouă furnizează o specificație a ceea ce clasele derivate trebuie să facă, interfața nu poate furniza nici un fel de detaliu de implementare sub formă de attribute sau de metode implementate. Consecința practică a acestui lucru este că derivarea din interfețe nu suferă de problemele potențiale pe care le are moștenirea multiplă, deoarece nu putem avea implementări diferite pentru aceeași metodă. Astfel, deși o clasă poate să extindă o singură clasă, ea poate să implementeze mai mult de o singură interfață.

5.5.1 Definirea unei interfețe

Din punct de vedere sintactic, nimic nu este mai simplu decât precizarea unei interfețe. Interfața arată ca o declarație a unei clase, doar că folosește cuvântul cheie `interface`. Ea constă dintr-o listă de metode care trebuie declarate. Un exemplu de interfață este `Comparable`, prezentată în **Listing 5.9**.

Interfața `Comparable` precizează două metode pe care orice clasă derivată din ea trebuie să le implementeze: `compareTo()` și `lessThan()`. Metoda `compareTo()` se va comporta similar cu metoda cu același nume din clasa `String`. Observați că nu este necesar să precizăm faptul că aceste metode sunt publice sau abstracte, deoarece acest lucru este implicit pentru metodele unei interfețe.

Listing 5.9: Interfața `Comparable`

```
1 public interface Comparable
2 {
3     int compareTo(Comparable rhs);
4     boolean lessThan(Comparable rhs);
5 }
```

5.5.2 Implementarea unei interfețe

O clasă implementează o interfață în doi pași:

1. declară că implementează interfața;
2. definește implementări pentru toate metodele din interfață.

Un exemplu este prezentat în clasa din **Listing 5.10**, în care se definește clasa `MyInteger`. Clasa `MyInteger` are un comportament asemănător cu al clasei `Integer`, din pachetul `java.lang`.

În linia 1 se observă că atunci când implementăm o interfață folosim cuvântul cheie `implements` în loc de `extends`. În această clasă putem scrie orice metode dorim, dar trebuie să definim cel puțin metodele din interfață. Interfața este implementată în liniile 27-36. Remarcați faptul că trebuie să implementăm exact metodele precizate în cadrul interfeței (`compareTo`, `lessThan`). Din acest motiv aceste metode au ca parametru un obiect de tip `Comparable` și nu un `MyInteger`.

O clasă care implementează o interfață poate fi extinsă cu condiția să nu fie finală. Astfel, dacă nu am fi declarat clasa `MyInteger` ca fiind finală, am fi putut-o extinde. O clasă care implementează o interfață poate totuși să extindă și o altă clasă. De exemplu, am fi putut, în principiu, scrie:

```
public class MyInteger extends Integer implements Comparable
```

Acest cod este incorect doar pentru că `Integer` este o clasă finală care nu poate fi astfel extinsă.

Listing 5.10: Clasa `MyInteger` care implementează interfața `Comparable`

```
1 final public class MyInteger implements Comparable
2 {
3     // constructor
4     public MyInteger(int value)
5     {
6         this.value = value;
7     }
8
9     // cateva metode
10    public String toString()
11    {
12        return Integer.toString(value);
13    }
14
15    public int intValue()
16    {
17        return value;
18    }
19
20    public boolean equals(Object rhs)
```

140


```

21 {
22     return rhs instanceof MyInteger &&
23         value == ((MyInteger) rhs).value;
24 }
25
26 //implementarea interfetei
27 public boolean lessThan(Comparable rhs)
28 {
29     return value < ((MyInteger) rhs).value;
30 }
31
32 public int compareTo(Comparable rhs)
33 {
34     return value < ((MyInteger) rhs).value ? -1 :
35         value == ((MyInteger) rhs).value ? 0 : 1;
36 }
37
38 private int value;
39 }

```

5.5.3 Interfețe multiple

Așa cum am menționat mai devreme, o clasă poate să implementeze mai mult de o singură interfață. Sintaxa pentru a realiza acest lucru este simplă. O clasă poate implementa mai multe interfețe prin:

1. precizarea interfețelor pe care le implementează;
2. implementarea tuturor metodelor din interfețe.

Interfața este cea mai abstractă clasă posibilă și reprezintă o soluție elegantă la problema moștenirii multiple.

5.6 Implementarea de componente generice

Să ne reamintim că unul dintre scopurile principale ale programării orientate pe obiecte este suportul pentru reutilizarea codului. Unul dintre mecanismele importante folosite pentru îndeplinirea acestui scop este *programarea generică*: dacă implementarea unei metode este identică pentru mai multe clase (cu excepția tipului de bază al obiectului), se poate folosi o *implementare generică* pentru a descrie funcționalitatea de bază. De exemplu, putem scrie o metodă care să sorteze un șir de elemente. *Algoritmul* pentru această metodă este independent de tipul de obiecte care sunt sortate, deci putem folosi un algoritm generic.

Spre deosebire de multe dintre limbajele de programare mai noi (cum ar fi C++) care utilizează *șabloane* (template) pentru a implementa programarea generică, Java nu oferă suport pentru implementarea directă a programării generice, deoarece programarea generică poate fi implementată folosind doar conceptele de bază ale moștenirii. În această secțiune vom prezenta cum pot fi implementate metode și clase generice în Java folosind principiile de bază ale moștenirii.

Ideea de bază în Java este că putem implementa o clasă generică folosind o superclasă adecvată, cum ar fi `Object`. În Java, dacă o clasă nu extinde o altă clasă, atunci ea extinde implicit clasa `Object` (din pachetul `java.lang`). Ca o consecință, fiecare clasă este o subclasă a lui `Object`.

Să considerăm clasa `MemoryCell` din **Listing 5.11**. Această clasă poate să rețină un obiect de tip `Object`. Deoarece `Object` este clasă de bază pentru orice clasă din Java, rezultă că această clasă poate să stocheze orice fel de obiecte.

Listing 5.11: Clasa generică `MemoryCell`

```

1 public class MemoryCell
2 {
3     private Object storedValue;
4
5     public Object read()
6     {
7         return storedValue;
8     }
9
10    public void write(Object x)
11    {
12        storedValue = x;
13    }
14 }

```

Există două detalii care trebuie luate în considerare atunci când folosim această strategie. Ambele sunt ilustrate în **Listing 5.12**. Funcția `main()` scrie valoarea 5 într-un obiect `MemoryCell`, după care citește din obiectul `MemoryCell`. În primul rând, tipurile primitive nu sunt obiecte. Astfel, `m.write(5)` ar fi fost incorect. Totuși, aceasta nu este o problemă, deoarece Java dispune de clase *wrapper* (de "împachetare") pentru cele opt tipuri primitive. În loc să stocăm direct numărul 5, am stocat un obiect de tip `Integer` cu valoarea 5.

Al doilea detaliu este că rezultatul lui `m.read()` este un `Object`. Deoarece în clasa `Object` este definită o metodă `toString()`, nu este necesar să facem conversia de la `Object` la `Integer`. Referința returnată de

`m.read()` (de tip `Object`) este polimorfică și ea referă de fapt un obiect de tip `Integer`. În consecință, se va apela automat metoda `toString()` a clasei `Integer`.

Dacă am fi vrut totuși să extragem valoarea reținută în obiectul de tip `MemoryCell`, ar fi trebuit să scriem o linie de genul

```
int i = ((Integer) m.read()).intValue();
```

în care se convertește mai întâi valoarea returnată de `read` la `Integer`, după care se folosește metoda `intValue()` pentru a obține un `int`.

Deoarece clasele *wrapper* sunt clase finale, constructorul și accesorul `intValue()` pot fi expandate *inline*³ de către compilator, generând astfel un cod la fel de eficient ca utilizarea directă a unui `int`.

Listing 5.12: Folosirea clasei generice `MemoryCell`

```
1 public class TestMemoryCell
2 {
3     public static void main(String[] args)
4     {
5         MemoryCell m = new MemoryCell();
6
7         m.write(new MyInteger(5));
8         System.out.println("Continutul este: " + m.read());
9     }
10 }
```

Un al doilea exemplu este problema sortării. Anterior am prezentat o metodă `insertionSort()` (Listing 5.8) care lucrează cu un șir de clase `Shape`. Ar fi interesant să rescriem această metodă pentru a putea sorta un șir generic. Listing 5.13 prezintă o metodă de sortare generică `insertionSort()` care este practic identică cu metoda de sortare din Listing 5.8 doar că folosește `Comparable` în loc de `Shape`. Putem pune această metodă într-o clasă `Sort`. Observați că nu sortăm `Object`, ci `Comparable`. Metoda `insertionSort()` sortează un șir de elemente `Comparable`, deoarece folosește `lessThan()`. Aceasta înseamnă că doar clasele care implementează interfața `Comparable` pot fi sortate astfel. De remarcat faptul că `insertionSort()` nu poate să sorteze un șir de obiecte `Shape`, deoarece clasa `Shape` din Listing 5.4 nu implementează interfața `Comparable`. Unul dintre exerciții propune modificarea clasei `Shape` în acest sens.

Pentru a vedea cum poate fi folosită metoda generică de sortare vom scrie un program, prezentat în Listing 5.14, care citește un număr nelimitat de va-

³Deși anumiți termeni (de exemplu, *wrapper*, *inline*) sunt în limba engleză, am optat pentru păstrarea lor în formă originală, deoarece așa s-au consacrat în literatura de specialitate. Considerăm că traducerea în limba română ar fi inoportună, deteriorând semnificația lor.

lori întregi, le sortează și afișează rezultatul. Metoda `readIntArray()` a clasei `Reader` (prezentată în capitolul precedent) este folosită pentru a citi un șir de numere întregi. Transformăm apoi șirul citit într-un șir de elemente `MyInteger`, care implementează interfața `Comparable`. În linia 15 creăm șirul, iar în liniile 16-19 obiectele care sunt stocate în șir. Sortarea este realizată în linia 22. În final, afișăm rezultatele în liniile 26-29. De reținut că metoda `toString()` este implicit apelată pentru clasa `MyInteger`.

Listing 5.13: Algoritm de sortare generic

```

1 //sortare prin insertie
2 public static void insertionSort(Comparable[] a)
3 {
4     for(int p = 1; p < a.length; ++p)
5     {
6         Comparable tmp = a[p];
7         int j = p;
8         for ( ; j > 0 && tmp.lessThan(a[j-1]); --j)
9         {
10             a[j] = a[j-1];
11         }
12         a[j] = tmp;
13     }
14 }

```

Listing 5.14: Citirea unui șir de numere întregi și ordonarea lui folosind un algoritm generic

```

1 import io.*; //pachet definit anterior
2 public class SortIns
3 {
4     //program de test care citește numere întregi
5     //(cate unul pe linie), le ordonează și apoi le afișează
6
7     public static void main(String[] args)
8     {
9
10        //citește un șir de întregi
11        System.out.print("Elementele sirului: ");
12        int [] sir = Reader.readIntArray();
13
14        //conversie la un șir de MyInteger
15        MyInteger[] sirNou = new MyInteger[sir.length];
16        for (int i = 0; i < sir.length; ++i)
17        {
18            sirNou[i] = new MyInteger(sir[i]);
19        }
20
21        //aplica metoda de sortare

```

```

22     Sort.insertionSort(sirNou);
23
24     //afiseaza rezultatul sortarii
25     System.out.println("Rezultatul sortarii: ");
26     for(int i = 0; i < sirNou.length; ++i)
27     {
28         System.out.println(sirNou[i]);
29     }
30 }
31 }

```

5.7 Clase interioare (inner classes)

Clasele interioare reprezintă un tip special de clase care, după cum le spune și numele, sunt definite în interiorul altor clase. Clasele care includ clase interioare se numesc (evident) clase exterioare. Clasele interioare sunt o facilitare importantă a limbajului Java, deoarece ne permit să grupăm clasele care aparțin în mod logic una de celalaltă și să controlăm nivelul de vizibilitate între aceste clase. Clasele interioare reprezintă un concept distinct de compoziția claselor (o clasă are un atribut de tipul altei clase). Diferențele le veți înțelege aprofundând treptat noțiunea de clasă interioară.

Necesitatea claselor interioare nu este deloc evidentă (cinstit vorbind, câți dintre programatorii de Java care citesc această lucrare au auzit sau au folosit clase interioare?). Sperăm ca exemplul din această carte să clarifice utilitatea lor.

Așa cum era de așteptat, o clasă interioară se definește în cadrul altei clase care o cuprinde:

```

1 /**
2  * Clasa exterioara.
3  */
4 public class Scrisoare
5 {
6     class Destinatie
7     {
8         ...
9     }
10
11     //alte posibile clase interioare
12     ...
13
14     //atribute si metode ale clasei exterioare
15     ...
16 }

```

În exemplul de mai sus clasa exterioară este *Scrisoare*, iar cea interioară este *Destinatie*. Pentru a face acest exemplu funcțional, creați fișierul sursă *Scrisoare.java* ca în **Listing 5.15**.

Listing 5.15: Clasa *Scrisoare* care cuprinde clasa interioară *Destinatie*

```

1 /**
2  * Clasa exterioara Scrisoare
3  */
4  public class Scrisoare
5  {
6      /**
7       * Clasa interioara Destinatie.
8       */
9      class Destinatie
10     {
11         /** Destinatia unei scrisori.*/
12         private String dest;
13
14
15         /** Creaza o destinatie pentru scrisoare.*/
16         Destinatie(String dest)
17         {
18             this.dest = dest;
19         }
20
21         /** Obtine destinatia scrisorii.*/
22         public String obtineDestinatie()
23         {
24             return dest;
25         }
26     }
27
28     /** Trimite scrisoare la adresa specificata.*/
29     public void trimiteScrisoare(String dest)
30     {
31         Destinatie d = new Destinatie(dest);
32
33         System.out.println("Scrisoarea a fost trimisa " +
34             "la destinatia: " + d.obtineDestinatie());
35     }
36
37     /** Programul principal.*/
38     public static void main(String[] args)
39     {
40         //creaza o scrisoare
41         Scrisoare s = new Scrisoare();
42
43         //trimite scrisoarea in "SUA"
44         s.trimiteScrisoare("SUA");

```

```

45 }
46 }

```

Prin rularea acestui exemplu se va obține următorul rezultat:

Scrisoarea a fost trimisa la destinatia: SUA

Dacă priviți cu atenție rezultatul compilării fișierului `Scrisoare.java`, veți observa apariția a două fișiere: `Scrisoare.class` și `Scrisoare$Destinatie.class`. Cum rezultatul compilării fiecărei clase este un fișier cu extensia `.class`, care conține bytecode-ul pentru clasa respectivă, este normal ca și o clasă interioară să producă un fișier `.class`, care să conțină bytecode-ul produs de compilator. Pentru a face sesizabilă legătura dintre clasa exterioară și cea interioară chiar și în situația în care fișierele sursă ale claselor nu sunt disponibile, creatorii limbajului Java au ales ca fiecare clasă interioară să respecte o formulă strictă în ceea ce privește numele fișierului `.class` care îi corespunde. Această formulă arată astfel:

`NumeClasaExterioara$NumeClasaInterioara`

În exemplul nostru, clasa interioară `Destinatie`, care este utilizată în metoda `trimiteScrisoare()`, arată ca orice altă clasă. Singura diferență pare a fi faptul că este definită în interiorul altei clase. Vom vedea mai târziu că există și alte diferențe mai subtile.

În exemplul anterior clasa interioară `Destinatie` nu a fost utilizată în mod direct în programul principal, dar nimic nu ne împiedică să o utilizăm. Utilizarea în metoda `main` a clasei interioare se realizează astfel:

```

1 public static void main(String [] args)
2 {
3     //creaza o scrisoare
4     Scrisoare s = new Scrisoare();
5
6     //trimite scrisoarea in "SUA"
7     s.trimiteScrisoare("SUA");
8
9     //creaza o alta destinatie
10    Destinatie d = s.new Destinatie("Canada");
11 }

```

Sintaxa pentru crearea unei clase interioare este neobișnuită și poate pune în încurcătură chiar și programatorii care au ceva experiență în Java. După cum se poate observa, pentru a crea direct o instanță a clasei interioare (în exemplul nostru, `d`), este necesară utilizarea unei instanțe a clasei exterioare (în exemplul nostru, `s`). Astfel, nu este posibilă crearea unei instanțe a clasei interioare,

dacă nu este deja creată o instanță a clasei exterioare. Limbajul Java impune o astfel de strategie, deoarece obiectul interior (cu alte cuvinte, instanța clasei interioare) este conectat la obiectul exterior (instanța clasei exterioare) din care este creat. Această conexiune este specifică doar claselor interioare și va fi detaliată în cadrul acestui capitol.

Construcția inestetică de instanțiere a clasei interioare poate fi totuși evitată utilizând un mic artificiu. Vom adăuga în clasa exterioară *Scrisoare* o metodă *catre* care creează un obiect de tip destinație:

```

1 ....
2
3 /** Stabilește destinația unei scrisori.*/
4 public Destinație catre(String dest)
5 {
6     return new Destinație(dest);
7 }
8
9 ...
10
11 public static void main(String[] args)
12 {
13     ...
14
15     Destinație d = s.catre("Canada");
16 }

```

O clasă interioară poate fi accesată uneori și de alte clase, nu numai de către clasa exterioară care o conține. Pentru a defini o astfel de referință a clasei interioare *Destinație*, este necesar ca tipul referință asociat să conțină și numele clasei exterioare, în formatul *ClasaExterioara.ClasaInterioara*. Pentru exemplul nostru declararea ar arăta astfel:

```
Scrisoare.Destinație d;
```

Dacă necesitatea de a utiliza clasa interioară în cadrul altei clase apare prea frecvent, este un semn că am proiectat greșit aplicația, și că, probabil, clasa respectivă nu este cazul să fie interioară.

Să revenim pentru moment asupra conexiunii care se realizează între clasa interioară și cea exterioară. În momentul în care se crează o instanță a clasei interioare, se crează o legătură între clasa interioară respectivă și instanța clasei exterioare, care conține această clasă interioară. Acest lucru permite ca membrii clasei exterioare să fie accesibili clasei interioare, fără a folosi vreo calificare specială. De exemplu, putem modifica exemplul anterior, introducând un nou atribut pentru clasa exterioară, care va fi accesat în clasa interioară:

```
public class Scrisoare
```



```

2 {
3     ...
4
5     /**
6      * Pentru a verifica daca scrisoarea a fost trimisa
7      * la destinatie.
8      */
9     boolean trimisa;
10
11     ...
12
13     class Destinatie
14     {
15         ...
16
17         Destinatie(String dest)
18         {
19             this.dest = dest;
20
21             //initial scrisoarea nu a fost trimisa
22             trimisa = false;
23         }
24
25         ...
26     }
27 }

```

După cum se poate observa, atributul `trimisa` al clasei exterioare a fost utilizat în clasa interioară, fără nici o calificare specială, ca și cum ar fi fost un atribut al acesteia. Explicația este următoarea: clasa interioară păstrează referința clasei exterioare, responsabilă de crearea instanței clasei interioare (în exemplul nostru, această referință este `s`). Când un membru al clasei exterioare este utilizat în cadrul clasei interioare, referința aceasta (ascunsă) este folosită pentru a accesa acel membru. Din fericire pentru noi, compilatorul este cel care ține cont de toate aceste detalii. Ca urmare, compilatorul va considera drept eroare situația în care nu va putea accesa referința clasei exterioare.

O altă caracteristică specifică doar claselor interioare este că ele pot fi ascunse complet de orice clasă, indiferent de pachetul în care se află. Spre deosebire de acest caz, mecanismul standard de ascundere a claselor permite doar ca o clasă să fie definită "prietenosă" (friendly), ceea ce presupune că ea este totuși accesibilă claselor din același pachet. Așadar, dacă doriți ca implementarea unei clase să fie complet ascunsă celorlalte clase, indiferent de pachetul în care se află, puteți implementa clasa ca fiind interioară. Așa cum era de așteptat ascunderea unei clase interioare se face prin declararea ei ca fiind `private`:

```

1 private class Destinatie
2 {

```

```

3     ...
4 }

```

În acest moment, clasa `Destinatie` este inutilizabilă pentru orice altă clasă cu excepția clasei `Scrisoare` în care a fost definită. Pentru a verifica acest lucru, să mai adăugăm o clasă în fișierul `Scrisoare.java`:

```

1 public class Scrisoare
2 {
3     ...
4 }
5
6 /** Clasa de test pentru clasa interioara privata.*/
7 class Test()
8 {
9     public static void main(String[] args)
10    {
11        Scrisoare.Destinatie d; //EROARE DE COMPILARE
12    }
13 }

```

Încercarea de a compila acest program este sortită eșecului, deoarece clasa interioară `Destinatie` reprezintă un membru privat al clasei `Scrisoare` și nu poate fi accesat în afara clasei al cărei atribut este. Compilatorul va afișa o eroare de genul:

```

Scrisoare.java:59: Scrisoare.Destinatie has private
access in Scrisoare
        Scrisoare.Destinatie d;
                        ^
1 error

```

Deoarece clasa interioară este declarată ca fiind `private`, nici o clasă, cu excepția clasei exterioare `Scrisoare`, nu o poate accesa. Cu alte cuvinte utilizarea clasei interioare este restricționată total în afara clasei exterioare. Astfel, clasele interioare oferă o modalitate de a preveni orice dependență de alte clase, dar și de a ascunde complet detaliile de implementare.

Analog o clasă interioară poate fi declarată ca fiind `protected`, ceea ce înseamnă că doar clasa exterioară și clasele care o derivează pot accesa clasa interioară respectivă.

Un aspect important care trebuie reținut este următorul: clasele normale (care nu sunt interioare) *nu* pot fi declarate `private` sau `protected`, ci doar `public` sau *friendly* (fără modificador de acces). De altfel, nici nu ar avea sens o clasă ne-interioară declarată în acel mod, deoarece nu am putea defini față de cine este `private` sau `protected`.

5.7.1 Clasificarea claselor interioare

În general, clasele interioare pe care le veți folosi în programele dumneavoastră vor fi clase simple, ușor de înțeles și utilizat, asemănătoare cu cele create în exemplele anterioare. Totuși, limbajul Java oferă și alte tipuri de clase interioare, ceva mai complexe.

În total, limbajul Java oferă patru tipuri de clase interioare:

- clase membre statice ale clasei exterioare

Ca orice metodă statică a unei clase, o clasă membru static are acces la toate metodele și atributele statice ale clasei părinte (cu alte cuvinte, ale clasei exterioare);

- clase membre nestatice ale clasei exterioare

Spre deosebire de clasele membre statice, cele nestatice au acces la toate metodele și atributele clasei exterioare, inclusiv la referința `this` a acesteia. Clasele interioare utilizate în exemplele precedente sunt de acest tip;

- clase locale

Clasele locale sunt definite în cadrul unui bloc de cod, fiind vizibile doar în cadrul acelui bloc, similar cu o variabilă locală definită în cadrul unei metode;

- clase anonime

Clasele anonime sunt clase locale fără nume.

5.7.2 Clasele membre statice ale clasei exterioare

Conexiunea dintre referința clasei exterioare și cea a clasei interioare poate fi eliminată în situația în care clasa interioară este declarată statică. În cazul unei clase interioare statice:

- nu este necesară o instanță a clasei exterioare pentru a crea o instanță a clasei interioare;
- pot exista membri statici în clasa interioară respectivă.

Putem modifica exemplul nostru, pentru a utiliza clase interioare statice. Iată cum arată fișierul sursă `Scrisoare.java` în această situație:

Listing 5.16: Exemplu utilizare clase interioare statice

```

1 /**
2  * Clasa exterioara Scrisoare
3  */
4 public class Scrisoare
5 {
6     /**
7      * Clasa interioara Destinatie.
8      */
9     private static class Destinatie
10    {
11        /** Destinatia unei scrisori.*/
12        private String dest;
13
14
15        /** Creeaza o destinatie pentru scrisoare.*/
16        Destinatie(String dest)
17        {
18            this.dest = dest;
19        }
20
21        /** Obtine destinatia scrisorii.*/
22        public String obtineDestinatie()
23        {
24            return dest;
25        }
26
27        /** Exemplu de membru static al clasei statice.*/
28        public static void f()
29        {
30        }
31    }
32
33    public static Destinatie catre(String dest)
34    {
35        return new Destinatie(dest);
36    }
37
38    /** Trimite scrisoare la adresa specificata.*/
39    public static void trimiteScrisoare(Destinatie d)
40    {
41        System.out.println("Scrisoarea a fost trimisa" +
42            " la destinatia: " + d.obtineDestinatie());
43    }
44
45    /** Programul principal.*/
46    public static void main(String[] args)
47    {
48        Destinatie d = catre("Canada");
49    }

```

```

50         //trimite scrisoarea la destinatie
51         trimiteScrisoare(d);
52     }
53 }
54 }

```

Folosind clase statice interioare, nu a mai fost necesară crearea unei instanțe a clasei exterioare `Scrisoare`. După rularea programului se va afișa următorul rezultat:

Scrisoarea a fost trimisa la destinatia: Canada

5.7.3 Clasele membre nestatice ale clasei exterioare

Clasele membre nestatice au fost prezentate pe larg la începutul acestui subcapitol, motiv pentru care au mai rămas puține detalii de adăugat:

- spre deosebire de clasele interioare statice, cele nestatice nu pot avea membri statici;
- uneori este necesară referința clasei exterioare, care este obținută folosind sintaxa `NumeClasaExterioara.this`. În exemplul nostru, referința la clasa exterioară se obține prin `Scrisoare.this`.

5.7.4 Clase locale

O clasă locală poate fi creată în cadrul unei metode, ca în exemplul următor:

Listing 5.17: Exemplu utilizare clase locale

```

1 /**
2  * Clasa exterioara Scrisoare
3  */
4 public class Scrisoare
5 {
6     /** Metoda ce cuprinde o clasa interioara */
7     public String catre(String dest)
8     {
9         /**
10          * Clasa interioara Destinatie.
11          */
12         class Destinatie
13         {
14             /** Destinatia unei scrisori. */
15             private String dest;
16         }
17     }

```

```

18         /** Creaza o destinatie pentru scrisoare.*/
19         Destinatie(String dest)
20         {
21             this.dest = dest;
22         }
23
24         /** Obtine destinatia scrisorii.*/
25         public String obtineDestinatia()
26         {
27             return dest;
28         }
29     }
30
31     return (new Destinatie(dest)).obtineDestinatia();
32 }
33
34 /** Trimite scrisoare la adresa specificata.*/
35 public void trimiteScrisoare(String dest)
36 {
37     System.out.println(" Scrisoarea a fost trimisa" +
38         " la destinatia: " + catre(dest));
39 }
40
41 /** Programul principal.*/
42 public static void main(String[] args)
43 {
44     Scrisoare s = new Scrisoare();
45
46     //trimite scrisoarea la destinatie
47     s.trimiteScrisoare("Romania");
48
49 }
50 }

```

Este evident faptul că aplicația nu are o utilitate practică, din moment ce afișează chiar stringul pe care îl primește ca parametru. Ea reprezintă doar un pretext pentru a vedea cum se crează o clasă interioară locală. Este ușor de observat că metoda `catre` cuprinde clasa interioară `Destinatia` în interiorul ei. Fiind definită în această metodă, clasa `Destinatia` este accesibilă doar în cadrul metodei. Încercarea de a o utiliza în afara metodei respective produce o eroare la compilarea programului.

Prin rularea programului se obține următorul rezultat:

```
Scrisoarea a fost trimisa la destinatia: Romania
```

Pentru a complica și mai mult lucrurile, trebuie să adăugăm că o clasă interioară poate fi definită și în cadrul unui bloc de instrucțiuni, asemănător cu exemplul anterior, în care metoda `catre` este modificată astfel:

```

1 public String catre(String dest)
2 {
3     if(dest != null)
4     {
5         /**
6          * Clasa interioara Destinatie.
7          */
8         class Destinatie
9         {
10            /** Destinatia unei scrisori.*/
11            private String dest;
12
13
14            /** Creaza o destinatie pentru scrisoare.*/
15            Destinatie(String dest)
16            {
17                this.dest = dest;
18            }
19
20            /** Obtine destinatia scrisorii.*/
21            public String obtineDestinatia()
22            {
23                return dest;
24            }
25        }
26
27        return (new Destinatie(dest)).obteneDestinatia();
28    }
29    else
30    {
31        return null;
32    }
33 }

```

Deși această utilizare pare destul de curioasă, clasa interioară *Destinatie* poate fi definită în cadrul unui bloc de instrucțiuni (în cazul nostru, instrucțiunea *if*), fiind vizibilă doar în acel bloc. Pentru a risipi eventuale urme de îndoială este util de reținut că definirea clasei *Destinatie* în cadrul instrucțiunii *if*, nu înseamnă că această clasă este creată în funcție de îndeplinirea condiției din *if*. Clasa va fi creată ca orice altă clasă la compilarea programului, dar nu va putea fi utilizată decât în cadrul blocului de instrucțiuni în care a fost definită.

Clasele locale sunt foarte rar utilizate din cauza codului mai puțin lizibil pe care îl crează (noi nu le vom utiliza deloc în cadrul acestei lucrări). Totuși ele există și pot fi utilizate în situații în care programatorul consideră că este nevoie de ele.

5.7.5 Clase anonime

Dacă exemplele anterioare v-au creat impresia că noțiunea de clasă locală este mai "deosebită", pregătiți-vă să vă întăriți această convingere, urmărind cu atenție modul de definire al claselor anonime. Pentru o mai bună înțelegere, iată un exemplu ce definește o clasă anonimă:

Listing 5.18: Model de definire a unei clase anonime

```

1 /**
2  * Clasa exterioara Scrisoare
3  */
4 public class Scrisoare
5 {
6     public Destinatie trimite()
7     {
8         return new Destinatie()
9         {
10             private String dest2;
11
12             public String obtineDestinatia()
13             {
14                 return "Germania";
15             }
16         }; // ";" este necesar la definirea claselor anonime
17     }
18
19     /** Programul principal.*/
20     public static void main(String[] args)
21     {
22         Scrisoare s = new Scrisoare();
23
24         Destinatie d = s.trimite();
25     }
26 }
27
28 class Destinatie
29 {
30     private String dest;
31 }

```

Analizând exemplul întâlnim două clase, *Scrisoare* și *Destinatie*, și o modalitate mai puțin întâlnită de definire a metodei *trimite*. Această metodă combină procesul de returnare a unei instanțe a clasei *Destinatie*, la linia 8, cu cel de definire a unei clase fără nume (anonimă), la liniile 9-16. Clasa anonimă reprezintă o subclasă a clasei *Destinatie*, datorită faptului că definirea clasei apare imediat după numele clasei *Destinatie*. În consecință mai apare o clasă, despre care nu putem spune ce nume are, ci doar că extinde

clasa `Destinatie`.

Definirea metodei `trimite` poate fi analizată în două părți:

- crearea și returnarea unei instanțe a clasei `Destinatie`:

```
return new Destinatie();
```

- definirea clasei anonime:

```
private String dest2;

public String obtineDestinatia()
{
    return "Germania";
}
```

Această construcție trebuie privită ca o modalitate de a defini o clasă care derivatează clasa `Destinatie`, fără a mai specifica numele acestei clase. Mai precis, clasa anonimă este prescurtarea codului următor:

```
1 class DestinatiaMea extends Destinatie
2 {
3     private String dest2;
4
5     public String obtineDestinatia()
6     {
7         return "Germania";
8     }
9 }
10
11 return new DestinatiaMea();
```

Una dintre cele mai importante caracteristici ale claselor interioare anonime este aceea că nu pot avea constructori expliți. Este evidentă această concluzie din moment ce o astfel de clasă nu are un nume.

Clasa de bază `Destinatie` este însă o clasă exterioară (deci, neanonimă), care poate avea constructori de orice tip. Apelul unui constructor neimplicit (cu parametri) se poate realiza astfel:

```
1 public class Scrisoare
2 {
3     ...
4
5     public Destinatie trimite(String destinatie)
6     {
7         /** Utilizarea constructorului explicit. */
8         return new Destinatie(destinatie)
9     }
```

```

10         private String dest2;
11
12         public String obtineDestinatia()
13         {
14             return "";
15         }
16     };
17 }
18
19 ...
20 }
21
22 class Destinatie
23 {
24     private String dest;
25
26     Destinatie(String dest)
27     {
28         this.dest = dest;
29     }
30 }

```

Nici un exemplu de utilizare a claselor anonime, dintre cele prezentate până în acest moment, nu poate să execute operații de inițializare asupra atributelor clasei anonime, pentru a simula funcționalitatea unui constructor explicit. Pentru a realiza acest lucru este necesar ca parametrul cu care va fi inițializat atributul în cauză să fie declarat `final`, ca în exemplul:

```

1 //argumentul este declarat final.
2 public Destinatie trimite(final String destinatie)
3 {
4     return new Destinatie()
5     {
6         private String dest2 = destinatie;
7
8         public String obtineDestinatia()
9         {
10             return dest2;
11         }
12     };
13 }

```

Așa cum ați putut constata deja, clasele anonime crează un cod mai puțin lizibil, motiv pentru care este indicat ca folosirea lor să fie limitată la acele clase care sunt foarte mici ca dimensiune (o metodă sau două) și care au o utilizare foarte ușor de înțeles. Veți fi probabil surprinși să aflați că probabil cel mai des utilizate clase interioare în practică sunt cele anonime. Clasele anonime sunt preferate de programatori când este vorba de a defini clase simple derivate din

Thread (vezi capitolul 8), clase de tip `Listener` (utilizate foarte frecvent în AWT și Swing) sau alte clase având o singură metodă, deoarece codul clasei apare exact în locul în care este folosit, ușurând astfel înțelegerea programului. Clasele anonime mai au și avantajul că se pot aduce mici modificări claselor existente fără a încărca inutil aplicația cu o arborescență inutilă de clase triviale.

5.7.6 Avantajele și dezavantajele claselor interioare

Avantajele utilizării claselor interioare

Deși par incomode la prima vedere, clasele interioare nu reprezintă doar un simplu mecanism de "ascundere" a codului, prin plasarea unor clase în interiorul altor clase. Clasele interioare sunt capabile de mai mult, pentru că au posibilitatea de comunicare cu clasa exterioară.

În principiu, clasele interioare oferă următoarele tipuri de avantaje:

- avantajul orientării pe obiecte

Folosind clasele interioare codul devine chiar mai orientat pe obiecte, decât în lipsa lor, deoarece ele permit "eliminarea" anumitor părți de cod din cadrul unei clase (clasa exterioară) și introducerea lor într-o clasă proprie (clasa interioară). Altfel spus, din punct de vedere al orientării pe obiecte, funcționalitatea care nu aparține de clasa exterioară este înlăturată și plasată într-o clasă interioară, rezultând astfel o decuplare a funcționalităților clasei exterioare și un cod mult mai elegant și mai clar. Mai mult, clasa interioară poate avea în continuare acces la toate atributele și metodele clasei exterioare, în cazul în care este membru nestatic al clasei exterioare;

- avantajul organizării claselor

Clasele interioare permit o organizare mai bună a structurii pachetelor. Astfel, clasele care nu trebuie să fie accesibile altor clase, chiar și celor din același pachet, pot fi definite ca fiind clase interioare, în așa fel încât să fie accesate doar de clasa exterioară care le conține.

Dezavantajele utilizării claselor interioare

Clasele interioare prezintă dezavantaje ce nu trebuie neglijate. Din punctul de vedere al întreținerii aplicațiilor, programatorii Java neexperimentați pot considera clasele interioare foarte dificil de înțeles. Este o reacție normală deoarece este necesară o perioadă de timp pentru ca programatorii Java să se obișnuiască cu acest concept și să considere utilă folosirea lui. Un alt dezavantaj ce merită a

fi semnalat este că utilizarea claselor interioare produce o creștere a numărului de clase ale aplicației.

Concluzii

Clasele interioare reprezintă un concept sofisticat care este întâlnit în mai multe limbaje de programare obiect-orientate. Nivelul de complexitate mai ridicat decât al altor concepte ale limbajului Java, face ca acest tip de clase să fie mai puțin preferat de programatorii începători. Cu timpul însă, ei acumulează experiența necesară pentru a recunoaște situațiile în care se impune utilizarea claselor interioare. Din acest motiv, pentru cei aflați în faza de inițiere în limbajul Java este suficient dacă reușesc să se familiarizeze cu sintaxa și modul lor de utilizare.

5.8 Identificarea tipurilor de date în faza de execuție

Pentru fiecare obiect existent în cadrul unei aplicații, limbajul Java oferă posibilitatea de a afla informații cu privire la clasa a cărei instanță este acel obiect. Aceste informații pot fi colectate în faza de execuție a aplicației respective ("runtime").

Pentru a vă convinge că astfel de informații sunt necesare, vom considera un exemplu, care va arăta în mod explicit de ce uneori este nevoie să cunoaștem tipul de date al unui obiect în faza de execuție a programului. Să presupunem că avem clasa de bază *Masina*, din care se derivează alte clase, câte una pentru fiecare tip de mașină. Astfel, avem clasele *Dacia*, *Opel*, *Ferrari*, etc. Să presupunem că toate mașinile au un preț, și că pentru fiecare mașină există un mod diferit de a-i calcula costul. În Java, aceasta înseamnă că metoda *afiseazaPret()* este o metodă abstractă a clasei de bază *Masina*, care are câte o implementare diferită pentru fiecare dintre clasele *Dacia*, *Opel*, *Ferrari*.

Listing 5.19: Clasa abstractă *Masina* și clasele derivate din ea

```
1 // Continutul fisierului Identificare.java
2 import java.util.*;
3
4 abstract class Masina
5 {
6     abstract void afiseazaPret();
7 }
8
```

```

9 class Dacia extends Masina
10 {
11     void afiseazaPret ()
12     {
13         System.out.println("Pretul unei Dacii este de " +
14             "5.000 de dolari.");
15     }
16 }
17
18 class Opel extends Masina
19 {
20     void afiseazaPret ()
21     {
22         System.out.println("Pretul unui Opel este de " +
23             "15.000 de dolari.");
24     }
25 }
26
27 class Ferrari extends Masina
28 {
29     void afiseazaPret ()
30     {
31         System.out.println("Pretul unui Ferrari este de " +
32             "75.000 de dolari.");
33     }
34 }
35
36 public class Identificare
37 {
38     public static void main(String[] args)
39     {
40         Vector v = new Vector();
41
42         v.add(new Dacia());
43         v.add(new Opel());
44         v.add(new Ferrari());
45
46         for(int i = 0 ; i < v.size(); i++)
47         {
48             ((Masina) v.elementAt(i)).afiseazaPret();
49         }
50     }
51 }

```

După cum reiese din **Listing 5.19**, fiecare clasă care extinde clasa de bază `Masina` are propria ei definiție a metodei `afiseazaPret()`. Metoda `main()` a programului principal crează un vector de elemente, în care se stochează câte un obiect, instanță a unei clase derivate din clasa de bază `Masina`. Deoarece clasa `Vector` stochează doar instanțe ale clasei `Object`, fiecare dintre in-

stanțele anterioare va fi stocată ca un obiect de tip `Object`, pierzându-se (aparent) informația specifică tipului a cărei instanță este de fapt. În final, vectorul este parcurs și pentru fiecare element al său este apelată metoda `afiseazaPret()`. Operația de *cast* (conversia explicită la tipul `Masina`) este necesară pentru a putea apela metoda `afiseazaPret()`, care nu există în clasa `Object`, redând astfel o parte din funcționalitatea obiectelor. Funcționalitatea specifică nu este redată în totalitate deoarece în acest moment se cunoaște doar faptul că vectorul conține instanțe ale claselor derivate din clasa `Masina`, fără a cunoaște exact aceste clase (`Dacia`, `Opel`, etc.). Totuși, după cum am văzut în paragraful 5.2.3, apelul metodei `afiseazaPret()` pe un obiect convertit la clasa `Masina`, va conduce prin polimorfism la apelul metodei cu același nume din clasa care a fost utilizată la crearea instanței (`Dacia`, `Opel`, sau `Ferrari`), obținându-se următorul rezultat:

```
Pretul unei Dacii este de 5.000 de dolari.
Pretul unui Opel este de 15.000 de dolari.
Pretul unui Ferrari este de 75.000 de dolari.
```

Astfel, folosind polimorfismul se pot crea instanțe ale unor clase specifice (`Dacia`, `Opel`, `Ferrari`), care sunt convertite apoi la o clasă de bază (`Masina`), ceea ce implică pierderea tipului specific al obiectului, după care, pe parcursul programului, se pot utiliza aceste referințe către clasa de bază. Efectul este că metoda exactă care va fi apelată pentru un obiect convertit la clasa `Masina` este determinată de referință, care poate fi către una din clasele: `Dacia`, `Opel`, `Ferrari`.

În general, exact acest lucru se urmărește: ca secvențele de cod să dețină cât mai puține informații despre tipul specific al unui obiect și să utilizeze doar reprezentarea generală a unei familii de obiecte (în cazul nostru, mașini). Ceea ce rezultă este un cod sursă mult mai ușor de scris, citit, înțeles și modificat. Este motivul pentru care polimorfismul este unul dintre principiile esențiale ale programării orientate pe obiecte (și totuși, atât de neglijate de începători).

Există însă situații în care este utilă chiar cunoașterea *exactă* a tipului unui obiect. De exemplu, dacă doriți să specificați că o mașină `Ferrari` este disponibilă doar în culoarea roșie, este necesar să testăm obiectul respectiv, pentru a vedea ce clasă instanțiază.

În Java există două metode de a afla tipul de date al unui obiect:

- identificarea standard a tipului de date, ce presupune că toate tipurile de date sunt disponibile atât în faza de compilare cât și în cea de execuție a aplicației;

- mecanismul de reflecție, care descoperă informații despre o clasă doar în timpul fazei de execuție.

5.8.1 Identificarea standard a tipurilor de date

În Java, identificarea standard a tipurilor de date se poate realiza în trei moduri:

- Utilizând operatorul de *cast* (în exemplul anterior acesta a fost utilizat la conversia către clasa *Masina*). Dacă operația de *cast* este incorectă (dacă obiectul nu poate fi convertit la tipul specificat), atunci va fi aruncată o excepție de tipul *ClassCastException*;
- Utilizând un obiect de tip *Class*, care conține informații despre clasa respectivă. Uneori clasa *Class* mai este denumită și *metaclasă*. Mașina virtuală Java atașează *fiecărei* clase existente într-un program un obiect de tipul *Class*, care este utilizat pentru a crea instanțe ale clasei respective, dar care poate fi utilizat și de programator pentru a afla date despre acea clasă. Există două posibilități de a obține obiectul de tip *Class* asociat unei clase (pentru exemplificare am ales clasa *Identificare* din programul anterior):

– folosind metoda *forName* a clasei *Class*:

```
1 Class c = null;
2 try
3 {
4     c = Class.forName("Identificare");
5 }
6 catch (ClassNotFoundException cnfe)
7 {
8 }
```

– folosind atributul *.class* asociat oricărei clase:

```
Class c = Identificare.class;
```

În general se preferă utilizarea celei de-a doua modalități, din mai multe motive. În primul rând este mai simplă, deoarece nu necesită prinderea și tratarea excepției *ClassNotFoundException*. În al doilea rând este mai sigură, deoarece existența clasei *Identificare* este verificată în faza de compilare și nu în cea de execuție, ca în primul caz.

Obiectul de tip `Class` asociat unei clase poate fi obținut și cu ajutorul unei instanțe a clasei respective, utilizând metoda `getClass()` a clasei `Object`:

```
Masina m = new Dacia();
Class c = m.getClass();
```

- Utilizând cuvântul cheie `instanceof`, care verifică dacă un obiect este instanță a unei clase specificate și returnează o valoare booleană. Iată un exemplu de utilizare în contextul aplicației anterioare:

```
1 Masina m = new Dacia();
2
3 if (m instanceof Dacia)
4 {
5     System.out.println("Masina aceasta este o Dacie.");
6 }
7 else
8 {
9     System.out.println("Masina aceasta NU este o Dacie.");
10 }
```

Rezultatul execuției acestei secvențe este:

Masina aceasta este o Dacie.

Limbajul Java oferă o alternativă pentru operatorul `instanceof`: metoda `isInstance()` a clasei `Class`. Această metodă are un avantaj față de operatorul `instanceof`: poate fi utilizată și pe obiecte de tipul `Class`. De exemplu, este greșită o secvență de cod de tipul următor:

```
1 Class[] c = { Dacia.class, Opel.class, Ferrari.class };
2
3 Masina m = new Dacia();
4
5 for (int i = 0; i < c.length; i++)
6 {
7     if (m instanceof c[i]) //EROARE
8     {
9         System.out.println("m este instanta a clasei " +
10                             c[i].getName());
11     }
12 }
```

Eroarea este datorată faptului că operatorul `instanceof` nu poate fi folosit decât în conjuncție cu numele clasei (`Dacia`), și nu cu un obiect de tipul `Class` (`Dacia.class`). Pentru a corecta această eroare, se

utilizează metoda `isInstance()`, înlocuind condiția din instrucțiunea `if`, cu condiția `c[i].isInstance(m)`. Rezultatul execuției programului astfel modificat este:

```
m este instanta a clasei Dacia
```

După cum se poate observa, metoda `isInstance()` oferă un mod dinamic de a apela operatorul `instanceof`. Ambele variante produc însă rezultate echivalente.

Există o diferență importantă între operatorul `instanceof` (implicit și metoda `isInstance()`), pentru că sunt operații cu rezultat echivalent) și compararea folosind obiecte de tipul `Class`. Diferența este vizibilă la aflarea informațiilor despre clase. Operatorul `instanceof` verifică dacă un obiect este o instanță a unei clase specificate sau a unei clase derivate din clasa specificată, în timp ce compararea obiectelor de tipul `Class` nu ține cont de moștenire, ci doar verifică strict dacă obiectul respectiv este o instanță a clasei specificate sau nu. Pentru a verifica afirmațiile anterioare, iată un exemplu:

```
1 Masina m = new Dacia();
2
3 System.out.println("m instanceof Masina " +
4                     (m instanceof Masina));
5 System.out.println("m instanceof Dacia " +
6                     (m instanceof Dacia));
7
8 System.out.println("m.getClass() == Masina.class " +
9                     (m.getClass() == Masina.class));
10 System.out.println("m.getClass() == Dacia.class " +
11                     (m.getClass() == Dacia.class));
```

Secvența afișează următorul rezultat:

```
m instanceof Masina true
m instanceof Dacia true
m.getClass() == Masina.class false
m.getClass() == Dacia.class true
```

5.8.2 Mecanismul de reflecție ("reflection")

Acest paragraf prezintă concepte mai avansate, care nu sunt necesare pentru înțelegerea informației din restul lucrării. Dacă nu sunteți direct interesați de

conceptul de *reflection*, puteți sări deocamdată peste el și veți putea reveni ulterior, când bagajul de cunoștințe acumulat vă va permite o înțelegere mai ușoară a lui.

Pentru a putea afla tipul unui anumit obiect puteți utiliza oricare dintre metodele anterioare, dar trebuie să aveți în vedere că există o anumită limitare. Dezavantajul constă în faptul că tipurile de date trebuie să fie cunoscute în faza de compilare, chiar dacă ele sunt utilizate în faza de execuție. Cu alte cuvinte, compilatorul trebuie să cunoască toate clasele care sunt utilizate pentru identificarea tipurilor de date în faza de execuție a programului.

La prima vedere nu pare a fi o limitare, dar poate deveni în situația în care este necesară instanțierea unei clase care nu este disponibilă mașinii virtuale JVM în momentul compilării. Aceasta sună probabil în acest moment a science-fiction, dar în aplicațiile concrete (în special cele client-server) această situație nu constituie o raritate. De exemplu, într-un fișier putem avea un șir de octeți care știm că reprezintă o clasă. În faza de compilare, compilatorul nu poate afla nici o informație legată de acel șir de octeți, deci utilizarea unei astfel de clase pare imposibilă. Din fericire, platforma Java oferă o soluție pentru această problemă: mecanismul de reflecție (*reflection*). Reflecția este un mecanism prin care se pot determina informații despre o clasă (numele clasei, numele metodelor, numele constructorilor, etc.) fără ca acea clasă să fie cunoscută compilatorului în faza de compilare.

Clasa `Class` prezentată în paragraful anterior suportă conceptul de reflecție, dar majoritatea claselor specializate pentru acest mecanism se găsesc în pachetul `java.lang.reflect`. Câteva dintre aceste clase, `Constructor`, `Field`, `Method`, vor fi prezentate mai pe larg pe parcursul acestui subcapitol.

În general, mecanismul de reflecție este utilizat în mod direct destul de rar de programatori, dar există situații în care sunt de folos anumite informații legate de o clasă. De aceea vom prezenta, cu ajutorul unor exemple simple, cum se pot afla informații utile despre o clasă.

Mediul de programare Java are la dispoziție un API destul de bogat pentru a asigura funcționalitatea mecanismului de reflecție. API-ul reprezintă clasele, obiectele și interfețele din mașina virtuală JVM și permite operații de tipul:

- determinarea clasei a cărei instanță este un anumit obiect;
- obținerea de informații despre modificatorii de acces ai clasei, despre attribute, metode, constructori și superclase;
- obținerea de informații despre constantele și metodele declarate într-o interfață;

- instanțierea unei clase, al cărei nume este cunoscut doar în faza de execuție;
- modificarea valorii unui atribut al unei clase, chiar dacă numele atributului este cunoscut doar în faza de execuție;
- apelarea unei metode a unei clase, chiar dacă numele acestei metode este cunoscut doar în faza de execuție.

Obținerea de informații despre o clasă

Pentru a afla informații despre o clasă trebuie să obținem obiectul de tipul `Class` asociat clasei respective, pentru că acel obiect deține toate informațiile legate de clasă. Folosind acest obiect se pot apela diverse metode ale clasei `Class`, care returnează obiecte de tipul `Constructor`, `Field`, `Method`, corespunzătoare constructorilor, atributelor și respectiv metodelor definite în cadrul clasei. De asemenea, un obiect `Class` poate reprezenta chiar și o interfață, situație în care se pot afla informații despre constantele definite, metodele declarate etc. Evident, nu toate metodele clasei `Class` sunt potrivite într-o astfel de situație. De exemplu, nu este normal apelul metodei `getConstructors`, deoarece o interfață nu are constructori.

Așadar, primul pas care trebuie realizat pentru o obține informații despre o clasă este obținerea obiectului de tipul `Class` asociat clasei respective. Principalele metode de a realiza acest demers au fost prezentate de-a lungul acestui subcapitol, dar este util să le reamintim:

- dacă există o instanță a clasei respective, se utilizează metoda `getClass`, prezentă în orice clasă Java (care extinde clasa `Object`):

```
Ferrari f = new Ferrari();
Class c = f.getClass();
```

- dacă numele clasei este disponibil în faza de compilare, se utilizează sufixul `.class` atașat numelui clasei:

```
Class c = Ferrari.class;
```

- dacă numele clasei nu este disponibil în faza de compilare, dar devine disponibil în faza de execuție, atunci se folosește metoda `forName` a clasei `Class`:

```
Class c = Class.forName("Ferrari");
```

Dacă obiectul de tip `Class` a fost obținut, pe baza lui se pot determina alte informații utile legate de clasa respectivă:

- Numele clasei, utilizând o secvență de cod asemănătoare cu:

```
1 Ferrari f = new Ferrari();
2 Class c = f.getClass();
3
4 //determina numele clasei
5 String name = c.getName();
6
7 //afiseaza acest nume
8 System.out.println(name);
```

Execuția secvenței anterioare are ca rezultat afișarea stringului "Ferrari".

- Tipul de modificador al clasei (public, protected, abstract, final, etc.)⁴:

```
1 Ferrari f = new Ferrari();
2 Class c = f.getClass();
3
4 int m = c.getModifiers();
5
6 if (Modifier.isPublic(m))
7 {
8     System.out.println("public");
9 }
10
11 if (Modifier.isFinal(m))
12 {
13     System.out.println("final");
14 }
```

Secvența nu va afișa nimic, deoarece clasa `Ferrari` nu este declarată nici `public`, nici `final`.

- Obiectul de tip `Class` asociat clasei de bază din care a fost derivată clasa respectivă (dacă este cazul):

```
1 Ferrari f = new Ferrari();
2 Class c = f.getClass();
3 Class sc = c.getSuperclass();
4 System.out.println(sc.getName());
```

⁴Pentru ca secvența să funcționeze trebuie importate clasele din pachetul `java.lang.reflect`. Analog și pentru restul exemplelor.

Secvența va afișa stringul "Masina".

- Obiectele de tip `Class` asociate interfețelor pe care le implementează clasa respectivă (dacă este cazul):

```
...
Class [] ci = c.getInterfaces();
...
```

- Dacă este clasă sau interfață:

```
1 Class c;
2 ...
3 if (c.isInterface())
4 {
5     ...
6 }
7 ...
```

- Informații despre atributele clasei, utilizând metoda `getFields()` a clasei `Class` (se pot determina numele, tipul, modificatorii de acces ai fiecărui atribut, sau chiar se pot da valori acestor atribute):

```
1 Class c = Ferrari.class;
2 Field [] fields = c.getFields();
3 for (int i = 0; i < fields.length; i++)
4 {
5     String fieldName = fields[i].getName();
6     String fieldType = (fields[i].getType()).getName();
7
8     System.out.println(fieldName + " " + fieldType);
9 }
```

- Informații despre constructorii unei clase, utilizând metoda `getConstructors()` a clasei `Class` (se pot determina numele, lista parametrilor constructorului, etc.):

```
1 Class c = Ferrari.class;
2 Constructor [] constr = c.getConstructors();
3 for (int i = 0; i < constr.length; i++)
4 {
5     String name = constr[i].getName();
6     ...
7 }
8 ...
```

- Informații despre metodele clasei, utilizând metoda `getMethods()` a clasei `Class` (se pot determina numele, tipul valorii returnate, lista argumentelor metodei, sau se poate apela metoda în sine folosind metoda `invoke()` a clasei `Method`):

```

1 Class c = Ferrari.class;
2 Method[] methods = c.getMethods();
3 for (int i = 0; i < methods.length; i++)
4 {
5     String name = methods[i].getName();
6     ...
7 }
8 ...

```

Obiectul de tip `Class` poate fi utilizat nu numai pentru a interoga clasele și a afla diverse informații despre ele. Cu ajutorul lui se pot realiza și instanțieri ale claselor, modificări ale atributelor, sau apelări ale metodelor. Iată câteva exemple de acest tip:

- Crearea unei instanțe a unei clase se poate realiza prin intermediul metodei `newInstance()` a clasei `Class`, în două moduri, în funcție de numărul de argumente al constructorului:

– constructori fără argumente

```

1 Object o = null;
2 try
3 {
4     Class c = Class.forName("Ferrari");
5     o = c.newInstance();
6 }
7 catch (Exception e)
8 {
9 }

```

– constructori cu argumente

```

1 try
2 {
3     Class[] argsClasses = { String.class };
4     Object[] argsValues = { "12" };
5
6     Class c = Class.forName("java.lang.Integer");
7     Constructor constr = c.getConstructor(argsClasses);
8 }

```

```

9      Object i = constr.newInstance( argsValues );
10
11      System.out.println( i );
12  }
13  catch( Exception e )
14  {
15  }

```

Pentru o mai bună înțelegere a instanțierii prin *reflection* a unei clase cu ajutorul unui constructor cu parametri, vom prezenta mai amănunțit acest exemplu. Instrucțiunile folosite au ca rezultat crearea unui obiect de tip `Integer`, utilizând un constructor al acestei clase, care are ca parametru un obiect `String`. În esență, codul este echivalent cu secvența:

```

String s = "12";
Integer i = new Integer(s);

```

Pentru a realiza acest lucru, au fost create mai întâi șirul cu tipurile argumentelor (`String`) și șirul cu valorile argumentelor (`"12"`). Apoi, pornind de la obiectul `Class` atașat clasei `Integer`, s-a obținut acel constructor al clasei `Integer` care primește ca parametru un obiect `String`. În final, utilizând acest constructor s-a creat o instanță a clasei `Integer`, cu argumentele specificate. În urma execuției se afișează valoarea `"12"`.

O diferență importantă față de varianta apelului constructorului fără parametri este aceea că `newInstance()` aparține clasei `Constructor` și nu clasei `Class` ca în exemplul precedent.

- Obținerea valorii unui atribut dintr-o clasă se realizează cu ajutorul metodei `getField()`. Aceasta returnează un obiect de tipul `Field`, care conține datele despre atributul respectiv. Dacă tipul de dată al atributului este primitiv, atunci valoarea atributului se obține apelând una din metodele `getInt()`, `getFloat()`, `getBoolean()`, etc., corespunzător tipului primitiv de dată al atributului. Dacă atributul reprezintă un obiect, atunci trebuie utilizată metoda `get()` pentru a afla valoarea acestuia. Drept exemplu vom prezenta o secvență de cod ce obține valoarea unui atribut (pentru aceasta presupunem că avem definită o clasă `Audi`, ce conține atributul `model` de tip `String`):

```

1  try
2  {
3      Audi a = new Audi( "TT-Coupe" );
4      Class c = a.getClass();

```

```

5
6      //obține obiectul ce conține informații despre model
7      Field f = c.getField("model");
8      //obține valoarea atributului model pentru instanța a
9      String s = (String) f.get(a);
10
11      System.out.println("Model = " + s);
12 }
13 catch (Exception e)
14 {
15 }

```

- Similar metodelor `get()` (`getInt()`, `getFloat()`, `get()` etc.), clasa `Field` mai cuprinde și metode `set` (`setInt()`, `setFloat()`, `set()` etc.) pentru a putea modifica valoarea atributului, setându-o la noua valoare dorită (pentru simplitate vom considera aceleași ipoteze ca la exemplul anterior):

```

1 try
2 {
3     Audi a = new Audi("TT-Coupe");
4     Class c = a.getClass();
5
6     //obține obiectul ce conține informații despre model
7     Field f = c.getField("model");
8
9     //modificam valoarea atributului model în "A4"
10    //pentru instanța a
11    f.set(a, "A4");
12 }
13 catch (Exception e)
14 {
15 }

```

- Apelul unei metode oarecare a unei clase oarecare se poate realiza dinamic prin intermediul clasei `Method`. De exemplu, pentru a concatena două stringuri se poate utiliza metoda `concat()` a clasei `String`, ceea ce ar conduce, folosind apelul dinamic al metodei respective, la următoarea secvență de cod ce afișează în final mesajul "Limbaajul Java":

```

1 try
2 {
3     //obținerea obiectului Class asociat clasei String
4     Class c = String.class;
5
6     //definire tipuri și valori ptr. parametrii metodei
7     //ce va fi apelată
8     Class[] tipParam = { String.class };

```



```

9      Object[] valParam = { "Java" };
10
11      //obține obiectul Method ce conține informații legate
12      //de metoda concat() cu lista de parametri specificată
13      Method m = c.getMethod("concat", tipParam);
14
15      //apel dinamic metoda concat() pe obiectul "Limbaajul "
16      //echivalent cu: "Limbaajul ".concat("Java");
17      String s = (String) m.invoke("Limbaajul ", valParam);
18
19      //afisarea rezultatului concatenării celor 2 siruri
20      System.out.println(s);
21 }
22 catch (Exception e)
23 {
24 }

```

Exemplele prezentate de-a lungul acestui subcapitol denotă faptul că mecanismul de reflecție permite ca informațiile despre obiecte să fie complet determinate în faza de execuție, fără nici o intervenție în faza de compilare. Analizând comparativ identificarea standard a tipului de date și mecanismul de reflecție se poate spune că cea mai importantă diferență dintre cele două este că în primul caz compilatorul verifică în faza de compilare clasele, în timp ce în cel de-al doilea caz, clasele sunt verificate în timpul fazei de execuție.

Rezumat

Moștenirea este o caracteristică puternică, fiind esența programării orientate pe obiecte și a limbajului Java. Ea permite abstractizarea funcționalității în clase abstracte, pentru a deriva apoi din aceste clase, alte clase care implementează și măresc funcționalitatea de bază.

Cea mai abstractă clasă, care nu implementează nimic poate fi specificată utilizând o *interfață*. Metodele specificate de o interfață trebuie definite de clasa care o implementează.

Un alt tip special de clase sunt clasele interioare, care sunt clase definite în interiorul altor clase. Deși mai puțin utilizate decât celelalte tipuri de clase, ele oferă facilități specifice și au avantaje importante.

Uneori este necesar să identificăm tipul unui obiect în faza de execuție a programului. Deși această facilitate nu va fi utilizată în cadrul acestei lucrări, ea reprezintă un element *sine qua non* pentru programatorii care ajung să folosească tehnologii client-server cum ar fi CORBA sau EJB.

Capitolul următor este unul mult așteptat, prin prisma numărului de referiri de care a avut parte de-a lungul capitolelor anterioare. Este vorba despre “Tratarea excepțiilor”.

Noțiuni fundamentale

cast: operator unar prin care o clasă este convertită la altă clasă.

clasă abstractă: clasă care nu poate fi instanțiată, dar care înglobează funcționalitatea comună a claselor derivate din ea.

clasă anonimă: clasă locală care nu are nume.

clasă de bază: clasă din care se derivează/implementează alte clase.

clasă finală: clasă care nu mai poate fi derivată.

clasă interioară: clasă definită în interiorul altei clase.

clasă derivată: clasă complet nouă care moștenește funcționalitatea clasei de bază.

clasă locală: clasă interioară definită în interiorul unui bloc de cod.

clasă wrapper: clasă care oferă un obiect ce stochează un tip primitiv. De exemplu, Integer este o clasă wrapper peste tipul primitiv int.

Class: clasă specială ce conține informații despre o clasă Java.

constructor super: constructorul clasei de bază din care a fost derivată clasa de față.

extends: cuvânt cheie prin care se specifică faptul că o clasă este derivată din altă clasă.

implements: cuvânt cheie prin care se specifică faptul că o clasă implementează metodele expuse de o interfață.

instanceof: operator prin care se verifică dacă un obiect este o instanță a unei clase specificate.

interfață: tip special de clasă Java, care nu conține nici un detaliu de implementare.

mecanismul "reflection": permite obținerea de informații despre o clasă în timpul fazei de execuție a programului.

moștenire: proces prin care se poate deriva o clasă nouă dintr-o clasă de bază, fără a afecta implementarea clasei de bază. Prin aceasta se poate crea o ierarhie de clase.

moștenire multiplă: proces prin care o clasă este derivată din mai multe clase de bază. Acest lucru nu este permis în Java. Totuși există o alternativă: implementarea mai multor interfețe.

polimorfism: abilitatea unei variabile referință de a referi obiecte de tipuri diferite. Când se apelează o metodă pe această variabilă, se apelează de fapt metoda tipului de obiect referit în momentul respectiv.

subclasă: denumire a clasei derivate.

superclasă: denumire a clasei de bază.

Erori frecvente

1. Membrii `private` din clasa de bază nu sunt vizibili în clasa derivată.
2. Dacă nu se definește nici un constructor în clasa derivată, iar clasa de bază nu are constructor implicit, se obține o eroare la compilare.
3. Clasele abstracte nu pot fi instanțiate.
4. Dacă în clasa derivată uităm să implementăm o metodă definită abstractă în clasa de bază, atunci clasa derivată devine ea însăși abstractă, ca și superclasa ei. Aceasta va genera o eroare la compilare (evident, dacă nu declarăm clasa derivată ca fiind abstractă). Clasa derivată devine concretă doar în momentul în care a implementat metoda abstractă moștenită.
5. Metodele finale nu pot fi redefinite, iar clasele finale nu pot fi extinse.
6. Metodele statice folosesc legarea statică, chiar dacă sunt redefinite în clasele derivate.
7. În clasa derivată attributele moștenite de la clasa de bază ar trebui inițializate ca un agregat, folosind metoda `super`. Dacă acești membri sunt `public` sau `protected`, ei vor putea fi ulterior modificați separat.
8. Dacă o metodă generică întoarce o referință generică, atunci de obicei trebuie realizată o conversie de tip pentru a obține obiectul efectiv returnat.
9. Membrii claselor interioare anonime pot fi inițializați doar cu attribute `final` ale clasei exterioare.

Exerciții

Pe scurt

1. Care membri din clasa de bază pot fi folosiți în clasa derivată? Care membri devin publici pentru utilizatorii claselor derivate?
2. Ce este agregarea?
3. Explicați polimorfismul.
4. Explicați legarea dinamică. Când nu se folosește legarea dinamică?
5. Ce este o metodă `final`?
6. Care este diferența dintre o clasă finală și alte clase? Când se folosesc clasele `final`?
7. Ce este o metodă abstractă?
8. Ce este o clasă abstractă?
9. Ce este o interfață? Prin ce diferă o interfață de o clasă abstractă? Ce fel de membri poate conține o interfață?
10. Cum sunt implementate componentele generice în Java?
11. Ce este o clasă interioară și de câte tipuri sunt clasele interioare?
12. Prezentați pe scurt procesul de identificare a tipurilor de date în timpul fazei de execuție a programului.

În practică

1. Scrieți două metode generice `min()` și `max()`, fiecare acceptând doi parametri de tip `Comparable`. Folosiți metodele într-o clasă pe care o denumiți `MyInteger`.
2. Scrieți două metode generice `min()` și `max()`, fiecare acceptând un șir de `Comparable`. Folosiți apoi aceste metode pentru tipul `MyInteger`.
3. Adăugați a nouă clasă (de exemplu, `Triunghi`) la ierarhia de clase `Shape` și verificați prin intermediul metodei de sortare dacă polimorfismul funcționează ca și în cazul claselor celelalte.

4. Creați o ierarhie de clase *Felina*, care să conțină clasele *Pisica*, *Tigru*, *Leu*, *Leopard*, *Ghepard* etc. (pentru detalii consultați un atlas biologic...). Definiți în clasa de bază metode care sunt comune pentru toate felinele, cum ar fi `getNum()`, `getCuloare()`, și în fiecare clasă metode specifice fiecărei feline (de exemplu `toarce()`, sau `getSoarece()` în cazul clasei *Pisica*). Redefiniți unele din metodele clasei de bază în clasele derivate. Creați un șir de feline și apelați metodele din clasa de bază pentru a vedea ce se întâmplă.
5. Modificați exemplul anterior astfel încât clasa *Felina* să fie o clasă abstractă. Definiți metodele clasei *Felina* ca fiind abstracte ori de câte ori este posibil.
6. Creați o clasă abstractă fără nici o metodă abstractă și verificați faptul că nu o puteți instanția.
7. Modificați exercițiul 4 pentru a ilustra ordinea de construire a claselor de bază și a celor derivate prin afișarea de mesaje în constructori. Apoi adăugați membri de tip referință la fiecare clasă și urmăriți ordinea în care sunt inițializați acești membri.
8. Creați o clasă de bază cu două metode, astfel încât prima metodă să o apeleze pe cea de-a doua. Derivați o clasă și redefiniți cea de-a doua metodă. Creați acum o instanță a clasei derivate, convertiți-o folosind operatorul de cast la clasa de bază și apelați prima metodă. Explicați rezultatul.
9. Modificați clasa *Shape* astfel încât ea să poată fi folosită de către un algoritm de sortare generic.

Proiecte de programare

1. Rescrieți ierarhia de clase *Shape* pentru a reține aria ca un membru privat, care este calculat de către constructorul clasei *Shape*. Constructorii din clasele derivate trebuie să calculeze aria și să trimită rezultatul către metoda `super`. Faceți din `area()` o metodă finală care doar returnează valoarea acestui atribut.
2. Adăugați conceptul de poziție la ierarhia *Shape* prin includerea coordonatelor ca date membru. Adăugați apoi o metodă `distance`.
3. Scrieți o clasă abstractă *Date* din care să derivați clasa *GregorianCalendar* (care reprezintă o dată în formatul nostru obișnuit).

6. Tratarea excepțiilor

Dacă e verde, e biologie, dacă miroase urât e chimie, dacă are numere e matematică, dacă nu funcționează e tehnologie.

Autor necunoscut

Nici o cantitate de experimente nu îmi poate valida teoria. Un singur experiment este însă suficient pentru a o infirma.

Albert Einstein

În capitolul anterior am prezentat concepte importante ale programării orientate pe obiecte, cum ar fi moștenirea, polimorfismul, programarea generică și clasele abstracte. În acest capitol vom prezenta modul în care se pot trata elegant erorile care apar inevitabil în timpul execuției unui program folosind din plin principiile prezentate până acum.

Vom afla în curând:

- Ce constituie o situație de eroare în Java;
- Ce sunt excepțiile și de câte tipuri sunt ele;
- Care este ierarhia de excepții în limbajul Java;
- Cum sunt folosite excepțiile pentru a semnaliza situații de eroare.

6.1 Ce sunt excepțiile?

Programatorii cunosc faptul că erori apar în orice aplicație software, indiferent de limbajul în care aceasta a fost scrisă. Dar ce se întâmplă *după* ce aceste

erori apar? Este posibil ca programul să le trateze și să își reia mersul normal al execuției sau este pur și simplu forțat să se termine? Dacă este posibilă tratarea lor, atunci cum este ea realizată și de cine? Capitolul de față încearcă să răspundă tuturor acestor întrebări.

Java folosește noțiunea de excepție pentru a oferi posibilitatea de tratare a erorilor pentru programele scrise în acest limbaj. Excepția poate fi considerată ca un eveniment care are loc în timpul execuției unui program, prin care este afectată (întreruptă) secvența de execuție normală a instrucțiunilor din acel program. Excepțiile reprezintă modul în care Java indică o situație anormală (de eroare) semnalată în procesul de execuție a unei metode apelate.

Ca în orice alt limbaj de programare, în Java, erorile pot apare în ambele faze ale implementării unei aplicații software: compilare și execuție. Momentul ideal pentru a descoperi erorile dintr-un program este cel al compilării, înainte de a încerca execuția programului. Totuși, nu toate erorile pot fi descoperite în faza de compilare. Unele dintre ele nu pot fi detectate și, implicit, tratate decât în faza de execuție a programului (*run-time*). În Java, tratarea erorilor în faza de execuție a unui program este posibilă prin intermediul excepțiilor.

Există numeroase tipuri de erori care pot provoca situații de excepție, împiedicând astfel execuția normală a unui program. Aceste probleme pot varia de la defecțiuni serioase de *hardware*, cum ar fi distrugerea parțială a unui *hard-disk*, până la simple erori de programare, cum ar fi accesarea într-un șir a unui element de pe o poziție mai mare decât lungimea șirului. Când o eroare de acest tip are loc în cadrul unei metode Java, metoda respectivă crează un obiect de tip excepție pe care îl predă apoi sistemului, sau cu alte cuvinte, mașinii virtuale JVM. Rezultă de aici că excepțiile sunt și ele niște clase Java obișnuite. Obiectul de tip excepție care este predat sistemului conține informații despre excepția generată, incluzând tipul ei și starea programului în momentul în care eroarea a apărut. Sistemul este apoi responsabil pentru a găsi secvența de cod care tratează excepția întâlnită. În terminologia Java, crearea unei excepții și predarea ei către sistem poartă numele de *aruncare de excepții* (operația `throw`).

După ce excepția a fost predată sistemului, este sarcina acestuia de a descoperi codul care tratează excepția respectivă. Sistemul "știe" unde să caute acest cod, datorită următorului fapt: în Java, apelul unei metode este urmat de păstrarea sa în cadrul unei stive de apel (*call stack*). Astfel, dacă metoda `main()` a unui program Java apelează o metodă, să spunem `metodaA()`, atunci în stivă se adaugă metoda `main()` urmată de metoda `metodaA()`. Dacă metoda `metodaA()`, apelează o altă metodă `metodaB()`, atunci cea din urmă este adăugată și ea în stivă. Practic, în acest fel, se poate cunoaște pentru fiecare metodă în parte, de cine a fost apelată, prin parcurgerea stivei de apel de la vârf spre bază (pentru exemplul nostru, `metodaB()` a fost apelată

de `metodaA()`, care la rândul ei a fost apelată de metoda `main()`). Acest mod de a reține apelul unei metode este cheia soluției pe care mașina virtuală o utilizează în tratarea excepțiilor. Mașina virtuală caută în stiva de apel metoda care tratează excepția, pornind de la metoda în care a apărut eroarea, deci dinspre vârful stivei spre bază. Sistemul consideră căutarea încununată de succes în momentul în care găsește o metodă din stiva de apel care tratează o excepție de același tip cu excepția aruncată de metoda în care a apărut eroarea. Codul care tratează excepția respectivă reprezintă, în terminologia Java, codul care *prinde* excepția (operația `catch`). Metoda în care poate apare acest cod, diferă de la un caz la altul. Excepția poate fi prinsă chiar în metoda în care a fost generată, sau, dimpotrivă, în metodele predecesoare din stiva de apel. Ca exemplu, dacă o excepție a fost aruncată în metoda `B`, ea poate fi prinsă tot în metoda `B`, sau în metoda `A`, sau în metoda `main`. În majoritatea cazurilor de excepții este *obligatoriu* ca excepția aruncată să fie prinsă în cadrul uneia din metodele din stiva de apel. Există însă anumite cazuri, care vor fi prezentate pe parcursul acestui subcapitol, în care se poate întâmpla ca o excepție să fie aruncată în cadrul unei metode, dar să nu fie prinsă în nici o altă metodă din stiva de apel. În acest caz, programul Java este oprit din execuție iar excepția este afișată la terminal.

În concluzie, aruncarea unei excepții poate fi văzută ca un semnal de alarmă, care indică sistemului că a apărut o problemă pe care trebuie să o trateze pentru ca execuția programului să poată continua. Semnalul este tras în speranța că una dintre metodele anterior apelate (cele din stiva de apel) va reacționa și va trata problema, iar programul își va putea continua astfel execuția.

6.2 Tipuri de excepții

În Java, excepțiile sunt obiecte. Când este aruncată o excepție, este aruncat de fapt un obiect, o instanță a unei clase de excepție. Nu orice tip de obiect poate fi însă aruncat ca excepție. Excepții pot fi doar obiectele care sunt instanțe ale unor clase derivate din clasa `Throwable`. Clasa `Throwable`, definită în pachetul `java.lang`, reprezintă clasa de bază pentru întreaga familie de clase de excepții.

Figura 6.1: Ierarhia claselor de excepții ale limbajului Java (pe scurt)

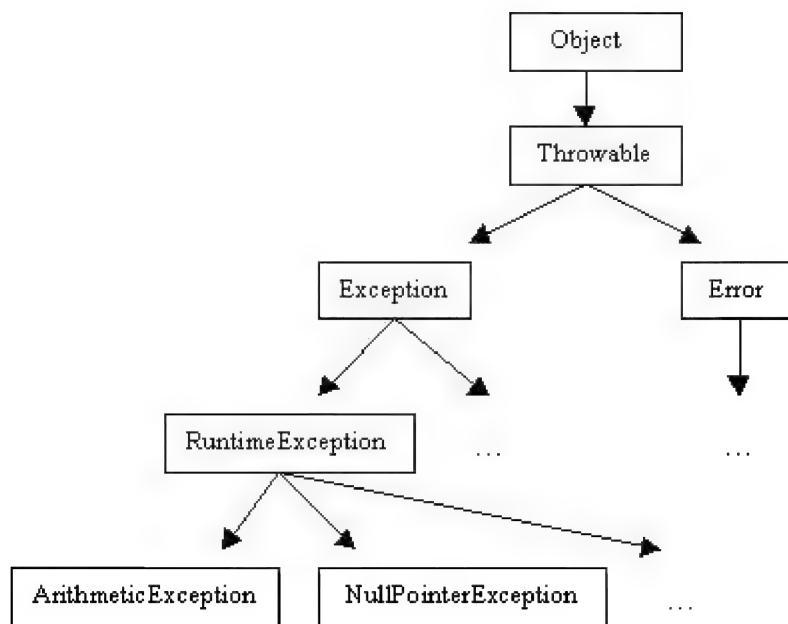


Figura 6.1 ilustrează ierarhia claselor de excepții, pornind de la clasa de bază `Throwable` și continuând cu câteva dintre subclasele cele mai importante. După cum se observă, clasa `Throwable` are doi descendenți direcți (clase derivate direct din clasa `Throwable`): `Error` și `Exception`.

Clasa `Error`, împreună cu toate clasele derivate din ea, descrie excepții grave, numite erori. Erorile reprezintă greșeli de compilare sau probleme mai serioase ale fazei de execuție a programului, probleme care în general nu pot fi tratate și necesită întreruperea execuției programului (de exemplu, `OutOfMemoryError`). În general, codul scris de programator nu ar trebui să arunce erori, ci doar excepții. De asemenea, programele Java obișnuite nu ar trebui să prindă erori, deoarece este puțin probabil ca erorile să poate fi în vreun fel tratate.

Clasa `Exception`, împreună cu toate clasele derivate din ea, reprezintă excepții. Excepțiile sunt aruncate pentru a semnala condiții anormale, care sunt deseori tratate de aplicație, deși există situații în care este posibil ca ex-

cepțiile respective să nu fie prinse și atunci aplicația va fi oprită din execuție. Cele mai multe programe aruncă și prind instanțe ale unor clase derivate din clasa `Exception`, de aceea `Exception` este clasa cea mai importantă pentru programator din acest punct de vedere. Clasa `Exception` are mulți descendenți definiți în cadrul pachetului `java.lang`. Aceste clase indică diverse tipuri de excepții care pot apărea în cadrul unui program Java. De exemplu, `NoSuchMethodException` indică faptul că programul a încercat să apeleze o metodă care nu există. Nu toate excepțiile sunt definite în pachetul `java.lang`. Unele dintre ele sunt create pentru a oferi suport pachetelor `net`, `util`, `io` etc. De exemplu, toate excepțiile I/O sunt derivate din `java.io.IOException` și sunt definite în pachetul `java.io`.

Există o subclasă a clasei `Exception`, care are o însemnătate specială în limbajul Java: `RuntimeException`. Clasa `RuntimeException` este clasa de bază pentru excepțiile care apar în timpul execuției unui program (excepții *runtime*). Pachetele Java definesc multiple clase pentru excepții *runtime*. Un exemplu des întâlnit este clasa `NullPointerException`, care apare în momentul în care se încearcă accesarea unui membru (atribut sau metodă) al unui obiect, prin intermediul unei referințe nule.

Costul verificării excepțiilor *runtime* depășește deseori avantajul de a le prinde, motiv pentru care compilatorul Java permite ca excepțiile *runtime* să nu fie prinse sau specificate. De asemenea, excepțiile *runtime* pot apărea practic în orice secvență de cod, motiv pentru care obligativitatea de a le prinde ar fi generat un cod greoi și obscur, umplut peste măsură cu secvențe de tratare a excepțiilor. Aceasta este cerința pentru ca o excepție să poată fi aruncată fără a fi prinsă: să fie instanță a unei clase derivate din `RuntimeException`.

Excepțiile se împart în două mari categorii:

- tratate (checked);
- netratate (unchecked).

Excepțiile *tratate* sunt denumite astfel pentru că atât compilatorul Java cât și mașina virtuală JVM verifică dacă excepțiile aruncate în cadrul unei metode sunt fie tratate în cadrul acelei metode, fie specificate ca fiind aruncate mai departe de metoda respectivă, urmând a fi tratate în alte metode.

O clasă de excepții este *tratată* sau *netratată*, în funcție de poziția ei în ierarhia claselor de excepții (cu clasa de bază `Throwable`). Potrivit acesteia, se consideră excepție *tratată*, orice clasă derivată din clasa `Exception`, inclusiv clasa `Exception`, mai puțin clasa `RuntimeException` și subclasele ei. Toate celelalte clase de excepții (clasa `Throwable`, clasa `Error` și subclasele

ei), împreună cu clasa `RuntimeException` și subclasele ei, reprezintă excepții *netratate*.

Diferența conceptuală dintre excepțiile tratate și cele netratate este că primele semnaleză situații anormale de care programatorul este obligat să țină seama. Cu alte cuvinte, aruncarea excepțiilor tratate în cadrul unei metode, obligă programatorul să le prindă și să le trateze mai devreme sau mai târziu în cadrul aplicației sale. Compilatorul este cel care îl obligă pe programator să trateze excepțiile respective. În cazul excepțiilor netratate, programatorul poate decide dacă prinde sau trece cu vederea excepția. De altfel, este posibil ca programatorul nici să nu știe că excepția netratată poate fi aruncată în cadrul metodei. Ca o consecință, este de la sine înțeles că nici compilatorul nu obligă programatorul să trateze excepția respectivă.

Utilizarea excepțiilor nu este nicidecum limitată doar la cele definite de platforma Java. Pe lângă aceste clase, programatorul poate defini propriile clase de excepții. Acest lucru este foarte important, pentru că deseori un programator simte nevoia să creeze propriile sale excepții, pentru a înfățișa o situație specială de eroare, pe care programul său este capabil să o producă, dar care nu a fost prevăzută la crearea ierarhiei de excepții. Această situație este perfect normală, deoarece creatorii limbajului Java nici nu ar fi putut să anticipeze toate situațiile de eroare ce pot apare în aplicații existente sau viitoare. Crearea propriilor excepții este chiar încurajată, deoarece regula generală este ca programul să fie cât mai precis. De aceea, programatorii nu trebuie să utilizeze excepții ambigue, de tipul `Exception`, ci subclase ale acestora, pentru a nuanța cât mai bine excepția apărută. De exemplu, dacă în cadrul programului nostru a fost citit un nume de utilizator invalid, atunci este de preferat să definim, să aruncăm și să prindem o excepție de genul `InvalidUserNameException`. În continuare, vor fi prezentate cele mai importante amănunte legate de crearea propriilor clase de excepții.

6.3 Definirea propriilor excepții de către programator

Java oferă două posibilități importante de a utiliza excepții într-un program:

- utilizarea unor excepții predefinite sau definite de alți programatori (de exemplu, clasele de excepții cuprinse în platforma Java) ;
- utilizarea unor excepții proprii, definite de programator.

Pachetele oferite de platforma Java oferă numeroase clase de excepții. Toate aceste clase sunt derivate direct sau indirect din clasa `Throwable` și permit programatorului să diferențieze diversele tipuri de excepții care pot să apară de-a lungul execuției unui program.

Platforma Java permite și crearea propriilor clase de excepții pentru a înfățișa situații problematice specifice care pot să apară în cadrul programelor. Deși necesitatea de a avea clase proprii de excepții poate să pară exagerată, totuși aceasta devine evidentă în cazul în care clasele scrise de programator vor fi reutilizate de către alți programatori. În astfel de situații, programatorul trebuie să își creeze propria structură de clase de excepții pentru a permite utilizatorilor claselor sale să diferențieze o eroare apărută în clasele sale, de erori apărute în clase scrise de alți programatori. Un alt motiv pentru care programatorii apelează la definirea de excepții proprii este nevoia de a avea o excepție care nu există în ierarhia de clase de excepții oferită de platforma Java, și care descrie o situație de excepție particulară aplicației respective.

Ca și clasele de excepții predefinite de platforma Java, clasele proprii de excepții trebuie să extindă (direct sau indirect) clasa `Throwable`. O primă tentație ar fi aceea de a extinde direct clasele proprii de excepție din clasa `Throwable`. La o privire mai atentă, se poate observa că `Throwable` are două subclase, `Error` și `Exception`, care împart în două erorile posibile dintr-un program Java. Marea majoritate a programelor Java utilizează excepții derivate din `Exception`, deoarece excepțiile de tip `Error` sunt rezervate pentru probleme mult mai complexe care au loc în cadrul sistemului (mașinii virtuale JVM). În definirea noilor excepțiilor, se consideră o bună practică adăugarea la numele clasei a sufixului `Exception` pentru toate clasele derivate (direct sau indirect) din `Exception` (de exemplu, `TeaTooHotException`). Evident, analog se procedează și în cazul (mult mai rar) al claselor derivate (direct sau indirect) din `Error`.

Deoarece clasele proprii de excepții sunt clase Java obișnuite, ele pot îngloba informații despre situația anormală care a condus la aruncarea lor. Astfel, se pot transmite informații utile către partea de program care tratează respectiva situație de eroare.

În general, clasele proprii de excepții sunt implementate astfel:

Listing 6.1: Clasa `TeaTooHotException`

```
1 public class TeaTooHotException extends Exception
2 {
3     public TeaTooHotException ()
4     {
5         super ();
6     }
}
```

184

```

7
8     public TeaTooHotException (String s)
9     {
10         super(s);
11     }
12 }

```

Clasa `TeaTooHotException` nu aduce practic nimic nou clasei `Exception`, deoarece definește doar doi constructori care apelează constructorii din clasa de bază. Totuși, *tipul* excepției în sine este foarte important: el ne ajută să discernem în momentul în care prindem excepția despre ce fel de eroare este vorba.

Folosind definirea anterioară, utilizatorul clasei de excepții va putea arunca o excepție în două moduri:

```

new TeaTooHotException ();

new TeaTooHotException ("Ceaiul este foarte fierbinte");

```

Este ușor de observat că în cel de al doilea caz situația de eroare transmite și o informație, sub forma mesajului "Ceaiul este foarte fierbinte", care poate fi folosit de codul care tratează excepția apărută pentru a afișa un mesaj informativ.

Informațiile transmise de excepții nu se limitează doar la mesaje de tip `String`. Dacă mesajul `String` nu este suficient pentru a îngloba întreaga informație dorită, se pot adăuga date noi clasei de excepții, sub forma atributelor și metodelor.

Listing 6.2 prezintă o variantă modificată a clasei `TeaTooHotException` care înglobează în informațiile transmise și temperatura ceaiului (engl. *tea*) din exemplul nostru.

Listing 6.2: Clasa `TeaTooHotException` (versiune modificată)

```

1 public class TeaTooHotException extends Exception
2 {
3     int temperature;
4
5     public TeaTooHotException ()
6     {
7         super();
8     }
9
10    public TeaTooHotException (String s)
11    {
12        super(s);
13    }

```

```
14
15 public TeaTooHotException (int temperature)
16 {
17     this.temperature = temperature;
18 }
19
20 public int getTemperature ()
21 {
22     return temperature;
23 }
24 }
```

Constructorul cu argument de tip `int` ne permite să creăm o excepție care înglobează ca informație temperatura ceaiului prea fierbinte (engl. *too hot*) pentru a fi consumat. Iată un exemplu de utilizare a acestui constructor:

```
new TeaTooHotException (70);
```

În momentul în care excepția este prinsă și tratată, undeva în cadrul aplicației, se vor putea obține informații suplimentare referitoare la temperatura care a provocat excepția `TeaTooHotException`, utilizând metoda `getTemperature()`.

În cele mai multe cazuri o asemenea abordare nu este necesară și singura informație (esențială) legată de excepție este tipul obiectului creat (cu alte cuvinte, numele clasei), fără alte informații adiacente stocate în cadrul obiectului respectiv.

6.4 Prinderea și tratarea excepțiilor

Când o metodă aruncă excepții, ea presupune că acestea sunt prinse și tratate undeva în cadrul aplicației. Pentru prinderea și tratarea lor, Java pune la dispoziția programatorilor trei cuvinte cheie, care vor fi descrise în continuare:

- `try`
- `catch`
- `finally`

6.4.1 Blocul `try`

Primul pas în prinderea și tratarea excepțiilor este acela de a "îngrădi" instrucțiunile care pot genera excepții într-un bloc de instrucțiuni, denumit bloc `try`. Blocul `try` cuprinde instrucțiunile care pot arunca excepții. În general, un bloc `try` arată astfel:

```

1 try
2 {
3     //instrucțiuni care pot genera excepții
4     instrucțiuni;
5 }

```

Blocul `try` ține sub observație toate instrucțiunile pe care le conține, în așteptarea unei excepții aruncate de una dintre respectivele instrucțiuni. El poate fi văzut ca un bloc de instrucțiuni care sunt "încercate" (`try` = a încerca), urmând ca, în cazul în care ele nu se pot executa cu succes, eroarea să fie tratată. Evident că, dacă instrucțiunile se execută fără a genera probleme, nu mai poate fi vorba de tratarea excepțiilor și aplicația se va executa normal, fără a ține cont de partea de tratare a excepțiilor. Acesta reprezintă un mare avantaj oferit de modul de gestionare a erorilor în limbajul Java. Astfel, programatorului îi este permis să se concentreze asupra problemei pe care încearcă să o rezolve, fără a-și pune problema eventualelor erori care pot fi generate de codul lui. Aceste erori sunt tratate ulterior. Din acest motiv, codul devine mult mai ușor de scris și de citit, deoarece scopul pentru care codul a fost scris nu se mai confundă cu verificarea eventualelor erori.

Unui bloc `try` i se asociază obligatoriu unul sau mai multe blocuri `catch` și, opțional, un bloc `finally`.

6.4.2 Blocul `catch`

Blocul `catch` reprezintă locul în care sunt prinse și tratate excepțiile. Blocurile `catch` nu pot exista independent, ele apar doar în asociere cu un bloc `try`:

```

1 try
2 {
3     ...
4 }
5 catch (...)
6 {
7     ...
8 }
9 catch (...)
10 {
11     ...
12 }
13 ...

```

Între blocul `try` și primul bloc `catch` nu pot exista instrucțiuni intermediare. Forma generală a unui bloc `catch` este următoarea:

```

1 catch (ClasaDeExcepții numeVariabila)

```

```

2 {
3     instrucțiuni;
4 }

```

După cum se poate observa, instrucțiunea `catch` necesită un singur parametru formal, notat `numeVariabila`. Modul de definire a instrucțiunii `catch` este foarte asemănător cu cel de definire a argumentelor unei metode Java, unde lista argumentelor unei metode era compusă din perechi de tipul *numeClasă* și *numeVariabila*, separate prin simbolul `,`. În cazul excepțiilor, `ClasaDeExceptii` reprezintă tipul de excepție care poate fi tratat, specificat prin numele clasei de excepție, derivată din `Throwable`. Parametrul formal, `numeVariabila`, reprezintă numele prin care poate fi referită excepția tratată de blocul `catch`. Excepția poate fi referită prin acest nume doar în cadrul blocului `catch`, pentru că durata de viață a variabilei `numeVariabila` se rezumă doar la blocul `catch`. Deseori variabila `numeVariabila` nu este utilizată, pentru că tipul clasei de excepție oferă îndeajuns de multă informație programatorului. Totuși, numele variabilei trebuie întotdeauna specificat, chiar dacă nu este utilizat niciodată în cadrul blocului `catch`. Pentru un exemplu de utilizare a blocurilor `try-catch`, să presupunem că încercăm accesarea unui element într-un șir:

```

1 int[] s = {4, 1, 8, 7};
2 try
3 {
4     System.out.println(s[6]);
5 }
6 catch (ArrayIndexOutOfBoundsException exc)
7 {
8     exc.printStackTrace();
9 }

```

Încercarea de a accesa un element care nu există în șirul `s` (acesta are 4 elemente, deci nu poate fi vorba de un element pe poziția 6) va arunca o excepție, `ArrayIndexOutOfBoundsException`. Aceasta este una dintre multiplele excepții predefinite de limbajul Java, derivată indirect din clasa `Throwable` și aflată în pachetul `java.lang`. Excepția astfel aruncată este prinsă de blocul `catch` asociat, care specifică faptul că tratează excepții `ArrayIndexOutOfBoundsException`. Faptul că excepția aruncată a fost prinsă de acest bloc `catch`, presupune că tot acest bloc o va și trata. Prin urmare, se execută instrucțiunile cuprinse în cadrul blocului `catch`, care sunt create pentru a trata cazul de eroare apărut. Excepția apărută este referită în cadrul blocului `catch` prin intermediul numelui `exc`. Variabila `exc` este o instanță a clasei `ArrayIndexOutOfBoundsException`, care este un obiect Java cu metode și atribute. Deoarece toate clasele de excepții sunt derivate din clasa

Throwable, ele moștenesc câteva metode utile pentru aflarea de informații despre situația de eroare. Printre aceste metode se află `printStackTrace()`, `getMessage()`, `toString()`, fiecare dintre ele afișând informații care diferă în cantitate și calitate de la un caz la altul. Pentru exemplul nostru, instrucțiunea `exc.printStackTrace()` va afișa următorul mesaj (`Exc.java` este numele fișierului care conține exemplul):

```
java.lang.ArrayIndexOutOfBoundsException
    at Exc.main(Exc.java:8)
```

Instrucțiunile din blocul `catch` se execută doar dacă instrucțiunile din blocul `try` generează excepții și aceste excepții sunt instanțe ale clasei predefinite `ArrayIndexOutOfBoundsException`. Este de la sine înțeles că dacă s-ar fi încercat accesarea unui element de pe o poziție mai mică decât lungimea șirului (de exemplu `s[2]`), nu ar mai fi fost o situație de eroare și, implicit, excepția `ArrayIndexOutOfBoundsException` nu ar mai fi fost aruncată, iar instrucțiunea din cadrul blocului `try` s-ar fi executat corect, afișând elementul de pe poziția 2 din șir (adică, 8).

Pentru ca un bloc `catch` să prindă și să trateze o excepție, nu este necesar ca tipul de excepție tratat de blocul `catch` și tipul de excepție aruncat de o metodă din blocul `try` să fie identice. Un bloc `catch`, care tratează un tip de excepții, va prinde și trata și clasele de excepții derivate din excepția respectivă. Cu alte cuvinte, în exemplul de față, nu este necesar să prindem excepția creată folosind clasa `ArrayIndexOutOfBoundsException`. Deoarece această clasă de excepții este derivată din `Exception`, codul ar putea arăta și așa:

```
1 int[] s = {4, 1, 8, 7};
2 try
3 {
4     System.out.println(s[6]);
5 }
6 catch(Exception exc)
7 {
8     exc.printStackTrace();
9 }
```

În această situație, blocul `catch` prinde toate excepțiile `Exception`, dar și clase de excepții derivate din clasa `Exception`, cum este și cazul clasei de excepții `ArrayIndexOutOfBoundsException`. Practic, în acest mod, se prind mai multe excepții cu același bloc `catch`, deoarece acest bloc prinde toate excepțiile de tip `Exception`, plus clasele de excepții derivate din aceasta. Dacă se folosește această abordare, de a avea un bloc `catch` pentru mai multe excepții generate, atunci codul de tratare a situației de eroare devine

prea general, pentru că excepția tratată de blocul `catch` nu este specializată, cu alte cuvinte este o clasă de bază pentru alte clase de excepții. Din această cauză, se preferă utilizarea blocurilor `catch` specializate, și nu a celor generale, pentru că ele pot oferi mai multe detalii despre situația de eroare. Totuși, cazul `Exception` este și el important, pentru că astfel se pot prinde și trata toate tipurile de excepții aruncate într-un mod unitar (`Exception` este clasa de bază a tuturor excepțiilor de programare).

Un alt aspect important care trebuie înțeles în legătură cu tratarea excepțiilor este că Java presupune că problema care a generat excepția este atât de critică, încât nu se poate realiza întoarcerea la instrucțiunea care a generat-o, cu alte cuvinte nu se poate reface situația dinaintea apariției erorii. Acest aspect este foarte important pentru cei care cred că tratarea unei excepții presupune realizarea unor operații care să îndrepte situația, iar apoi metoda care a generat-o să fie executată din nou, în speranța că a doua tentativă va fi încununată de succes. Java *nu* realizează acest lucru. După ce excepția a fost tratată, se execută în continuare următoarea linie de cod aflată după secvența `catch`. Totuși, se poate folosi un mic truc, care să ofere această posibilitate, și anume, instrucțiunea care generează eroarea să fie plasată în cadrul unui ciclu (`for`, `while`, `do`).

Este foarte posibil ca într-un bloc `try` să existe instrucțiuni care pot genera excepții de mai multe tipuri. Aceste excepții pot fi tratate separat, asociind blocului `try` mai multe blocuri `catch`. De exemplu, putem avea o metodă `drinkTea()`, care aruncă două excepții, `TeaTooHotException` și `TeaTooColdException`, ambele clase fiind derivate din `Exception`. În consecință, am putea avea următorul cod:

```

1 //temperatura ceaiului
2 int temperature = 25;
3
4 try
5 {
6     drinkTea(temperature);
7 }
8 catch (TeaTooHotException hotExc)
9 {
10    hotExc.printStackTrace();
11 }
12 catch (TeaTooColdException coldExc)
13 {
14    coldExc.printStackTrace();
15 }
```

Utilizarea mai multor blocuri `catch` asociate unui bloc `try` trebuie să țină cont de anumite reguli, care vor fi prezentate în continuare.

Blocurile `catch` sunt examinate în ordinea în care ele apar în fișierul sursă. În timpul procesului de examinare, primul bloc care este în măsură să trateze eroarea, o va face, iar celelalte blocuri nu vor mai fi luate în considerare, deși ele ar putea, la rândul lor, să trateze respectiva excepție. De aceea, ordinea blocurilor `catch` contează. Totodată, un bloc care tratează o excepție generală nu se poate afla înaintea unui bloc care tratează o excepție specializată, derivată din cea generală, deoarece secvența de tratare specializată nu ajunge să fie executată niciodată. Pentru exemplul nostru, putem considera excepția generală `Exception`, iar excepția specializată ca fiind `TeaTooHotException`, care este derivată din excepția generală. Ordinea blocurilor `catch` este obligatoriu următoarea:

```

1 //temperatura ceaiului
2 int temperature = 25;
3
4 try
5 {
6     drinkTea(temperature);
7 }
8 catch (TeaTooHotException hotExc)
9 {
10     hotExc.printStackTrace();
11 }
12 catch (Exception exc)
13 {
14     exc.printStackTrace();
15 }
```

Ordinea este logică, pentru că, dacă ar fi fost posibilă inversarea ordinii celor două excepții, atunci `Exception` ar fi fost tot timpul excepția prinsă, în timp ce `TeaTooHotException` ar fi fost imposibil de prins, deci codul de tratare a acestei excepții nu ar fi fost executat niciodată. Așadar, regula generală este că blocurile `catch` care tratează subclase de excepții trebuie să precedeze blocurile `catch` ce tratează superclase.

6.4.3 Blocul `finally`

Din momentul în care mașina virtuală JVM începe să execute un bloc `try`, ea poate avea mai multe posibilități de a ieși din acel bloc. Ar putea întâlni în cadrul blocului instrucțiuni de tipul `break`, `continue`, `return`, care să cauzeze ieșirea prematură din bloc. Pe de altă parte, diferitele excepții aruncate de codul din blocul `try` ar putea avea drept rezultat o altă întrerupere prematură a executării codului din acel bloc.

Există multe situații în care este de o importanță vitală să se execute anumite

instrucțiuni la ieșirea din blocul `try`, indiferent de modul în care această ieșire se realizează. De exemplu, dacă într-un bloc `try` este deschis un fișier pentru a scrie date în el, este de dorit o secvență de cod care să îl închidă, indiferent de cum se termină execuția blocului de scriere. Existența unei astfel de secvențe de cod este necesară și în situația în care se dorește eliberarea anumitor resurse utilizate în blocul `try`, cum ar fi fluxurile de date etc. Pentru a realiza acest deziderat, mediul de programare Java oferă noțiunea `finally`.

Clauza `finally` poate fi utilizată doar în asociere cu un bloc `try`. Ea constă, ca și blocul `try`, dintr-un bloc de instrucțiuni care permite ca o secvență de cod să fie executată, indiferent dacă blocul `try` aruncă sau nu excepții. Prin clauza `finally`, instrucțiunea `try-catch` devine completă, având următoarea sintaxă:

```
1 try
2 {
3     //instrucțiuni care pot arunca excepții
4     ...
5 }
6 catch (E1 e1)
7 {
8     //instrucțiuni care trateaza excepții de tipul E1
9     //și excepții derivate din clasa de excepții E1
10    ...
11 }
12 catch (E2 e2)
13 {
14     //instrucțiuni care trateaza excepții de tipul E2
15     //și excepții derivate din clasa de excepții E2
16     ...
17 } //numarul de blocuri catch este variabil
18 finally
19 {
20     //instrucțiuni care se executa indiferent de ce
21     //se intampla in blocul try
22     ...
23 }
```

Pentru o mai bună înțelegere a blocului `finally` să considerăm programul din **Listing 6.3**, în care avem o aplicație care deschide (sau crează dacă nu există) un fișier stocat pe hard-disk și scrie în el un șir de numere întregi.

Listing 6.3: Program de test pentru clauza `finally`

```
1 //Nume fisier : TestFinally.java
2
3 import java.io.*;
4
5 public class TestFinally
6 {
7     ...
8 }
9
10
11
12
```

```

6 {
7  public static void main(String[] args)
8  {
9      PrintWriter pw = null;
10     int[] s = {3, 7, 4, 6};
11     String msg = "";
12
13     try
14     {
15         msg = "Se incearca deschiderea fisierului...";
16         System.out.println(msg);
17         pw = new PrintWriter(new FileWriter(
18             "fisier.txt"));
19
20         for (int i = 0; i < s.length; i++)
21         {
22             pw.println(s[i]);
23         }
24     }
25     catch (IOException ioe)
26     {
27         msg = "Exceptie I/O prinsa: " + ioe.getMessage();
28         System.out.println(msg);
29     }
30     finally
31     {
32         if (pw != null)
33         {
34             pw.close();
35             System.out.println("Fisierul a fost inchis");
36         }
37         else
38         {
39             System.out.println("Fisierul nu a fost " +
40                 "deschis");
41         }
42     }
43 }
44 }

```

Programul afișează următoarele mesaje:

```

Se incearca deschiderea fisierului...
Fisierul a fost inchis

```

Totodată, programul crează un fișier numit `fisier.txt`, în același director cu fișierul sursă al aplicației noastre, cu următorul conținut:

3

7
4
6

La o analiză atentă se observă că instrucțiunile din blocul `try` nu au generat nici o excepție (nu se afișează mesajul "Excepție I/O prinsa: "), deci totul s-a executat fără eroare, iar ieșirea din blocul `try` nu a fost bruscă. Totuși, instrucțiunile din blocul `finally` s-au executat pentru că a fost afișat mesajul "Fișierul a fost închis".

Să încercăm acum să înlocuim numele fișierului, `fișier.txt`, cu o cale invalidă, cum ar fi `C:\test\fișier.txt`, cu condiția ca directorul `test` să NU existe pe partiția C: a hard-disk-ului dumneavoastră. Rulând programul astfel modificat, se obțin următoarele rezultate:

```
Se incearca deschiderea fișierului...
Excepție I/O prinsa: c:\test\fișier.txt (The system
cannot find the path specified)
Fișierul nu a fost deschis
```

În această situație, constructorul clasei `FileWriter` aruncă o excepție de tipul `IOException`, pentru că nu poate găsi directorul `test`, și în consecință nu poate crea fișierul `fișier.txt`. Excepția aruncată este tratată de blocul `catch` existent, care afișează mesajul "Excepție I/O...", iar apoi, se execută din nou instrucțiunile din blocul `finally`, afișându-se mesajul "Fișierul nu a fost deschis".

După cum se poate observa, blocul `finally` a fost executat indiferent dacă situația a fost de eroare (al doilea caz) sau nu (primul caz). Deoarece în blocul `finally` au fost eliberate resursele alocate în blocul `try` (în cazul nostru, fluxul de scriere `PrintWriter`), putem considera blocul `finally` un mecanism elegant de a "face curățenie" după execuția codului din blocul `try`.

Situația de eroare prezentată anterior nu este singulară. `IOException` pot fi generată și dacă directorul respectiv există (deci calea este validă), dar aplicația nu are drept de scriere în acel director, sau dacă nu mai există suficient spațiu pe disc pentru a scrie date în fișierul respectiv.

Cu siguranță că programatorii cu experiență în alte limbaje de programare (în special C++, unde această noțiune nu există) se vor întreba dacă existența clauzei `finally` este justificată. Nevoia de a avea clauza `finally` devine evidentă dacă analizăm mai atent programul anterior, în care, fără `finally`, fluxul de scriere ar fi putut fi eliberat doar dacă am fi plasat codul în fiecare bloc `catch` separat, ceea ce ar fi dus la o duplicare inutilă a codului.

6.5 Aruncarea excepțiilor

Înainte ca o excepție să fie prinsă, trebuie să existe, undeva în cadrul claselor utilizate, o secvență de cod care să arunce excepția respectivă. Excepțiile pot fi aruncate atât de codul scris de dumneavoastră, cât și codul din clase scrise de alți programatori (cum ar fi, excepțiile aruncate de clasele predefinite în platforma Java), sau chiar de mașina virtuală JVM. Indiferent de cine aruncă excepția, ea este întotdeauna aruncată folosind cuvântul cheie `throw`.

6.5.1 Instrucțiunea `throw`

Metodele Java aruncă excepții folosind instrucțiunea `throw`. Instrucțiunea `throw` este urmată de un singur argument, care trebuie să fie o instanță a unei clase derivate din clasa `Throwable`, și are următoarea formă:

```
throw instantaClasaDeExcepții ;
```

Iată câteva exemple de utilizare:

-

```
throw new Exception ();
```

-

```
Exception e = new Exception ();  
throw e;
```

-

```
//temperatura initiala a ceaiului  
int temperature = 30;  
  
//calculare intermediare  
....  
  
//consideram ca temperatura la care ceaiul  
//nu este inca fierbinte, este de 40 de grade  
if (temperature > 40)  
{  
    //ceaiul este foarte fierbinte si nu poate fi baut,  
    //deci suntem intr-o situatie de eroare,  
    //motiv pentru care aruncam o exceptie  
    throw new TeaTooHotException ();  
}
```

Încercarea de a arunca drept excepție un obiect care nu este instanță a unei clase de excepție, este semnalată drept eroare de către compilator.

În momentul aruncării unei excepții se realizează mai multe lucruri: mai întâi, se crează obiectul de tip excepție, în același mod în care se crează orice obiect Java, prin folosirea cuvântului cheie `new`. Apoi, se întrerupe execuția normală a instrucțiunilor, iar mecanismul de tratare a excepțiilor, aflat în JVM, preia sarcina de a căuta secvența de cod care tratează excepția respectivă, pentru a putea continua execuția programului.

Crearea obiectului de tip excepție constă de fapt în apelul unui constructor al clasei de excepție. Aceasta poate avea mai multe tipuri de constructori, mai ales dacă respectiva clasă de excepție este chiar o excepție proprie, scrisă de dumneavoastră (paragraful 6.3). În mod obișnuit sunt două tipuri de constructori care pot fi apelați: constructorul implicit, care nu are parametri, și constructorul cu un mesaj de tip `String`. De exemplu:

```
throw new TeaTooHotException ();
```

```
throw new TeaTooHotException ("Ceaiul este foarte " +  
                                "fierbinte.");
```

De obicei, se aruncă câte o clasă diferită de excepție pentru fiecare tip de eroare întâlnit. Având în vedere tipurile de constructori ai clasei de excepție, informația despre eroare este reprezentată atât în interiorul obiectului de excepție (cu ajutorul unui mesaj `String` etc.), cât și de tipul obiectului de excepție (cu alte cuvinte, de numele clasei de excepție).

Uneori este necesar ca o anumită excepție să fie rearuncată, adică să fie aruncată exact în locul în care a fost prinsă. Iată un exemplu de rearuncare a unei excepții:

```
1 try  
2 {  
3     ...  
4 }  
5 catch (Exception e)  
6 {  
7     throw e;  
8 }
```

Prinderea și rearuncarea aceleiași excepții (ca în exemplul anterior) nu are nici un sens, deoarece este ca și cum am ignora excepția respectivă. Situația este diferită însă când tipul excepției prinse nu este neapărat același cu tipul excepției rearuncate. Asemenea situații apar frecvent în aplicațiile ceva mai complexe în care o excepție de un anumit gen este "tradusă" într-o excepție de alt gen, dar cu conținut echivalent (de exemplu, într-o aplicație client-server, o excepție prin care serverul anunță că nu a putut procesa o anumită cerere a clientului,

este tradusă de programul client într-o altă excepție care să conțină un mesaj inteligibil pentru utilizator). În exemplul următor, `MyNewException` este o clasă excepție proprie aruncată în cazul prinderii unei excepții de intrare-ieșire:

```

1 try
2 {
3     ...
4 }
5 catch (IOException e)
6 {
7     throw new MyNewException();
8 }

```

6.5.2 Clauza `throws`

Orice metodă Java trebuie fie să prindă, fie să specifice clar toate excepțiile pe care le aruncă mai departe metodelor care o apelează. Este vorba, desigur, de excepțiile tratate (adică, cele care nu sunt derivate din `RuntimeException`, vezi paragraful 6.2) care pot fi aruncate de codul din interiorul metodei respective.

Am prezentat până acum doar cazul în care excepțiile sunt prinse și tratate în cadrul aceleași metode, nefiind transmise mai departe. Această secțiune va prezenta cea de a doua posibilitate: specificarea excepțiilor care nu sunt tratate în cadrul metodei respective, fiind aruncate mai departe, pentru a fi prinse și tratate de alte metode (cu alte cuvinte, vom învăța cum se aruncă pisica moartă în ograda vecinului).

Atunci când decidem ca o metodă să nu prindă o anumită excepție care poate fi aruncată de codul din interiorul ei, trebuie să precizăm faptul că metoda poate arunca la rândul ei excepția respectivă (deci vecinii trebuie avertizați în legătură cu posibilitatea de a se trezi cu o pisică decedată în ogradă). Această cerință este perfect explicabilă, pentru că specificarea excepțiilor aruncate mai departe de metodă ține de interfața metodei, cu alte cuvinte cei care utilizează o metodă trebuie să cunoască precis ce excepții aruncă chiar și în cazul în care nu au la dispoziție codul sursă al metodei, pentru a putea elabora o strategie de tratare a respectivelor excepții.

Specificarea excepțiilor aruncate de o metodă se realizează prin intermediul clauzei `throws`, utilizată în antetul metodei respective:

```

1 tipReturnat numeMetoda (listaArgumente) throws Exception1, ...
2 {
3     ...
4 }

```

Clauza `throws` se folosește în situația în care programatorul decide că nu este potrivit ca metoda în cauză să prindă și să trateze o excepție aruncată de o altă metodă pe care o apelează, sau dacă metoda aruncă propriile sale excepții.

Lista excepțiilor specificate în clauza `throws` cuprinde excepțiile aruncate direct, folosind instrucțiunea `throw`, precum și excepțiile (netratate) aruncate de alte metode apelate. Evident, ambele tipuri de excepții specificate în clauza `throws`, nu trebuie să fie tratate în cadrul metodei. Dacă acest lucru se întâmplă, atunci excepțiile nu mai pot fi precizate în clauza `throws`.

Iată un exemplu de specificare a excepțiilor care nu sunt prinse și tratate în cadrul unei metode:

```
1 public void drinkTea(int temp) throws TeaTooHotException
2 {
3     //daca temperatura e prea mare
4     if (temp > 40)
5     {
6         throw new TeaTooHotException();
7     }
8 }
```

Exemplul de față specifică faptul că metoda `drinkTea()` aruncă o excepție proprie, `TeaTooHotException`, dacă temperatura ceaiului este mai mare de 40 de grade. Datorită acestui lucru, apelul metodei `drinkTea()` într-o altă metodă se poate realiza doar dacă excepția `TeaTooHotException` este prinsă și tratată în cadrul acelei metode, sau dacă și acea metodă alege să arunce excepția mai departe, specificând-o în clauza `throws`:

•

```
1 public void serveCustomer()
2 {
3     int temperature = 45;
4
5     try
6     {
7         drinkTea(temperature);
8     }
9     catch (TeaTooHotException exc)
10    {
11        ;
12    }
13 }
```

•

```
1 public void serveCustomer() throws TeaTooHotException
2 {
```

```

3   int temperature = 45;
4
5   drinkTea( temperature );
6 }

```

Este evident că în cel de-al doilea caz, excepția va trebui prinsă și tratată la un moment dat, în cadrul unei metode. Deoarece este o excepție tratată, ea nu poate fi aruncată mai departe de fiecare metodă, fără să existe în final o metodă care să o prindă și să o trateze.

Specificarea excepțiilor aruncate de o metodă nu se poate evita, dacă metoda respectivă generează excepții pe care nu le tratează, deoarece compilatorul detectează această "scăpare" și o raportează drept eroare de compilare, propunând două soluții: tratarea lor în cadrul metodei sau specificarea lor în clauza `throws`.

Totuși există o posibilitate de a "păcăli" compilatorul, atunci când se specifică faptul că o metodă aruncă o anumită excepție, deși acest lucru nu se întâmplă. În această situație, compilatorul obligă apelurile metodei noastre să trateze excepția sau să o specifice în clauza `throws` a metodei apelante, chiar dacă metoda noastră nu aruncă propriu-zis excepția respectivă, ci doar pretinde că o aruncă. Acest lucru este util în momentul în care știm că în viitor metoda noastră va fi modificată astfel încât să arunce respectiva excepție, deoarece nu va mai fi necesară modificarea codului care apelează metoda, codul respectiv tratând excepția în avans, înainte ca ea să fie aruncată propriu-zis de metodă.

O ultimă observație: mecanismul de aruncare mai departe a excepțiilor oferă o modalitate deosebit de elegantă de a trata problemele care apar în decursul execuției la nivelul adecvat. De exemplu, este foarte posibil ca o metodă care citește un flux de octeți să întâlnească diverse probleme, cum ar fi: fluxul nu poate fi deschis, transmiterea de date s-a întrerupt prematur etc. Totuși, metoda respectivă este de un nivel prea jos pentru a putea lua decizii în legătură cu acțiunile care trebuie întreprinse în cazul unei erori (care pot chiar diferi de la o situație la alta). Aruncând o excepție, metoda noastră are posibilitatea de a anunța metodele de nivel mai înalt care au apelat-o de faptul că secvența de execuție nu a decurs normal, lăsând în seama acestora modul în care se tratează eroarea. Programatorii cu ceva experiență în C sau Pascal vor recunoaște probabil superioritatea acestei metode, în fața celei în care metoda care citea dintr-un flux de date seta o variabilă sau returna o anumită valoare care descria modul în care a decurs citirea din flux (de exemplu, returna 0 dacă citirea a decurs normal, 1 dacă nu s-a putut deschide fișierul, 2 dacă citirea s-a întrerupt etc.).

6.6 Sugestii pentru utilizarea eficientă a excepțiilor

Problemele cele mai mari cu care se confruntă programatorii atunci când utilizează excepțiile Java sunt *când* și *ce* excepții să arunce.

Pentru prima problemă, răspunsul sumar pe care un programator ar putea să-l ofere, ar arăta astfel:

Dacă o metodă întâlnește o situație anormală pe care nu o poate trata, atunci metoda respectivă ar trebui să arunce o excepție.

Din nefericire, acest răspuns este cât se poate de ambiguu, pentru că ne conduce către o altă întrebare: ce înseamnă o "situație anormală"? Aceasta poate fi considerată întrebarea cheie a tratării excepțiilor.

A decide dacă o anumită situație este anormală sau nu, rămâne însă o problemă subiectivă. Decizia nu este întotdeauna evidentă și de aceea depinde în mare măsură și de programator. Totuși, situația nu este chiar atât de gravă pe cât ați fi tentați să credeți. Pentru a veni în ajutorul programatorilor, creatorii limbajului Java au considerat că o bună regulă de utilizare a excepțiilor este aceea de a "evita utilizarea excepțiilor pentru a indica situații previzibile în funcționarea standard a unei metode".

În consecință, o situație anormală este un eveniment care nu poate fi prevăzut în cazul unei funcționări normale a metodei. Pentru o mai bună înțelegere a acestui concept, vom considera câteva exemple utile, în care se decide dacă este utilă sau nu utilizarea excepțiilor. Ca exemple am ales câteva situații întâlnite chiar de dezvoltatorii limbajului Java la crearea ierarhiei de clase oferită de platforma Java, pentru a vedea cum au fost acestea tratate (dacă au fost considerate situații anormale și, implicit, s-a recurs la utilizarea excepțiilor, sau dacă au fost considerate situații normale, previzibile).

Să considerăm mai întâi cazul metodei `read()` a clasei `FileInputStream`, disponibilă în pachetul `java.io`. Această metodă returnează un `byte` citit dintr-un flux de intrare specificat. Situația specială cu care s-au confruntat dezvoltatorii limbajului Java a fost următoarea: cum ar fi trebuit să trateze această metodă situația în care toate datele din fluxul de intrare au fost citite, deci s-a ajuns la sfârșitul fișierului? Ar fi trebuit ca metoda `read()` să arunce o excepție, pentru că s-a ajuns la finalul fișierului, sau, din contră, această situație să nu fie considerată specială, ci mai degrabă una previzibilă? Soluția aleasă de dezvoltatorii limbajului Java a fost că metoda *nu* trebuie să arunce o excepție, ci doar să returneze o valoare specială, `-1`, care să indice sfârșitul fișierului. Astfel, atingerea sfârșitului fișierului a fost considerată drept

o situația normală, previzibilă în utilizarea metodei `read()`. Motivul pentru care nu a fost considerată o situație anormală a fost acela că metoda obișnuită de a citi bytes dintr-un flux de intrare, este aceea de a încerca citirea lor pe rând, până când este atins finalul fișierului.

Al doilea caz este cel al metodei `readInt()`, clasa `DataInputStream`, din același pachet `java.io`. În momentul în care este apelată, această metodă citește patru bytes din fluxul de intrare și îi convertește într-un `int`. În această situație, creatorii limbajului Java au ales ca metoda `readInt()` să arunce o excepție, `EOFException`, atunci când se ajunge la sfârșitul unui fișier. Două au fost motivele pentru care această situație a fost considerată anormală:

- metoda `readInt()` nu poate returna o valoare specială pentru a semnaliza sfârșitul fișierului, pentru că toate valorile pe care le-ar putea returna sunt valori întregi care ar putea exista în acel fișier (cu alte cuvinte, `-1` nu ar putea fi considerat sfârșitul fișierului, pentru că ar putea exista un element `-1` în cadrul fișierului, și astfel am fi induși în eroare, deoarece prezența elementului `-1` în cadrul acelui fișier nu înseamnă obligatoriu că fișierul a fost parcurs până la final);
- metoda `readInt()` poate fi pusă în ipostaza de a putea citi doar unul, doi sau trei bytes din fluxul de intrare, și nu patru, cât îi este necesar pentru a forma un `int` (o variabilă de tip întreg este reprezentată în Java pe 4 octeți), ceea ce poate fi considerată o situație anormală. Datorită acestui fapt, excepția `EOFException` este definită ca o excepție tratată, motiv pentru care programatorii care utilizează metoda `readInt()` sunt obligați să prindă această excepție și să o trateze.

Al treilea exemplu se referă la interfața `java.util.Enumeration`, împreună cu cele două metode ale sale, `hasMoreElements()` și `nextElement()`. Această interfață reprezintă o modalitate de a parcurge o serie de elemente, câte unul pe rând, de la primul element până la ultimul element al seriei. Obținerea unui element din serie se face cu ajutorul metodei `nextElement()`, care returnează elementul curent din serie și mută poziția cursorului pe următorul element, care astfel devine noul element curent. Metoda `hasMoreElements()` returnează o valoare booleană, care indică dacă seria a fost parcursă în întregime sau nu s-a atins încă sfârșitul ei. Metoda `hasMoreElements()` trebuie să fie apelată de fiecare dată înaintea metodei `nextElement()` pentru a vedea dacă s-a ajuns la sfârșitul seriei de elemente sau nu. Această abordare ne arată că dezvoltatorii limbajului Java nu au considerat atingerea sfârșitului seriei de elemente ca fiind o situație anormală, ci una previzibilă. Totuși, dacă sfârșitul seriei a fost atins și este apelată metoda `nextElement()`, fără a apela

metoda `hasMoreElements()` în prealabil, atunci este aruncată o excepție, `NoSuchElementException`, care reprezintă o excepție netratată (subclasă a clasei `RuntimeException`), aruncată mai degrabă pentru a indica o eroare de programare (faptul că interfața nu este utilizată corect), decât pentru a indica sfârșitul seriei de elemente.

După cum se poate observa, regula general valabilă este că în cazul în care situația este previzibilă (de exemplu, sfârșitul unui fișier, al unei serii de elemente, sau obținerea unui rezultat nul la operația de căutare a unui element într-un șir de elemente), *nu* este recomandată utilizarea excepțiilor.

O altă abordare a problemei utilizării eficiente a excepțiilor este legată de modul în care metodele sunt utilizate. Să luăm cazul metodei `charAt()` a clasei `String`, din pachetul `java.lang`. Această metodă primește ca parametru un întreg și returnează caracterul din string aflat pe poziția indicată de parametru.

Dacă metoda `charAt()` este apelată cu parametrul `-1` sau un întreg de o valoare mai mare decât lungimea șirului, ea nu își poate executa sarcina pentru că datele sale de intrare sunt incorecte. Iar, dacă datele de intrare sunt incorecte, nici cele de ieșire (rezultatele) nu pot fi corecte. De aceea, metoda aruncă o excepție `StringIndexOutOfBoundsException`, care indică faptul că programatorul are o eroare de programare sau nu a utilizat clasa corect.

Pe de altă parte, dacă datele de intrare sunt corecte, dar dintr-un anumit motiv `charAt()` nu ar putea returna caracterul de pe poziția specificată, atunci metoda ar trebui să semnaleze acest lucru aruncând o excepție, prin care indică faptul că sunt probleme care necesită o tratare specifică.

Prin intermediul acestei abordări se poate trage concluzia că excepțiile trebuie utilizate atunci când metodele nu beneficiază de date de intrare corecte (precondiție încălcată) sau când, deși au date de intrare corecte, totuși, din anumite motive, nu își pot duce la bun sfârșit execuția (postcondiție încălcată).

Cea de a doua problemă majoră cu care se confruntă programatorii Java este *ce* excepții să arunce.

După ce s-a decis utilizarea excepțiilor, următorul pas este să decidem ce tipuri de excepții să utilizăm. Avem de ales între a arunca excepții instanțe ale clasei `Throwable` sau ale unei subclase a acesteia. Pe de altă parte, se pot arunca excepții predefinite de platforma Java (existente în Java API), sau putem crea propria ierarhie de excepții.

Prima sugestie în alegerea tipului de excepții este aceea de a arunca întotdeauna *excepții* (subclase ale clasei `Exception`) și nu *erori* (subclase ale clasei `Error`). Motivul este că `Error` are scopul de a semnaliza probleme complexe, delicate ale sistemului unde este preferabil să nu ne băgăm nasul.

A doua sugestie este legată de modul de utilizare a excepțiilor tratate (checked

exceptions) și a celor netratate (unchecked exceptions). După cum s-a observat de-a lungul acestui capitol, aruncarea în cadrul unei metode a unei excepții tratate, forțează codul care apelează metoda respectivă să trateze acea excepție. Pe de altă parte, aruncarea în cadrul unei metode a unei excepții netratate (excepții *runtime*), nu obligă codul care apelează metoda respectivă să trateze acea excepție, lăsând la latitudinea programatorului decizia de a prinde sau nu excepția respectivă.

Sugestia în această situație este că, dacă sunteți de părere că excepția trebuie obligatoriu tratată de utilizatorul codului dumneavoastră, atunci aruncați excepții tratate.

În general, excepțiile care indică o utilizare improprie a metodelor unei clase (o eroare de programare) sunt excepții netratate. Un exemplu potrivit este cel anterior, în care metoda `charAt()` aruncă excepția `StringIndexOutOfBoundsException`. Dacă metoda `charAt()` este apelată cu argumentul `-1`, ea aruncă o excepție prin care semnalează că nu a fost utilizată corect (nu există indicele `-1` într-un șir de elemente). Pe de altă parte, exemplul metodei `readInt()` din clasa `DataInputStream` și al excepției `EOFException` este util pentru a ilustra alegerea excepțiilor tratate. Deoarece excepția `EOFException` indică o eroare care a survenit în momentul citirii din fișier, nu poate fi vorba de o utilizare greșită a clasei `DataInputStream`. Excepția semnalează că, deși clasa a fost utilizată corect, totuși metoda nu a reușit să își îndeplinească funcționalitatea, ceea ce reprezintă o situație anormală, motiv pentru care programatorii care utilizează această metodă sunt obligați să trateze cazul de excepție.

Cea de-a treia și ultima sugestie este aceea de a înlocui clasele de excepții predefinite de platforma Java cu propriile clase de excepții, atunci când considerați că ar fi mai explicit astfel. Puteți realiza acest lucru dacă prindeți excepția predefinită Java, și, în codul de tratare a excepției respective, aruncați propria dumneavoastră excepție.

Utilizarea excepțiilor își are propriul ei preț din punct de vedere al resurselor utilizate. Datorită stivei de apel (call stack), creată de mașina virtuală JVM pentru a reține ordinea în care metodele au fost apelate, oprirea bruscă din execuție a unei metode (în caz de excepție) este mult mai costisitoare decât o execuție fără incidente a metodei respective, deoarece mașina virtuală consumă resurse căutând în stiva de apel, metoda care conține secvența de cod ce tratează respectiva excepție. Din acest motiv, excepțiile trebuie utilizate ponderat, mai ales în metodele care se apelează foarte frecvent și a căror viteză de execuție este esențială.

6.7 Avantajele utilizării excepțiilor. Concluzii

Existența unui mecanism de tratare a erorilor este una dintre principalele modalități de a crește robustețea unei aplicații. Tratarea erorilor este un principiu fundamental pentru orice aplicație, în special în Java, unde un țel foarte important este acela de a crea componente de aplicații care vor fi utilizate de alți programatori. Pentru a asigura dezvoltarea unei aplicații profesionale, fiecare dintre aceste componente trebuie să fie robustă.

Excepțiile oferă multe avantaje programatorilor. În primul rând, ele separă codul care tratează erorile de codul care funcționează normal. Codul suspect de a se executa cu eroare (chiar dacă această eroare poate apare doar în 0,001% din cazuri) este încadrat într-un bloc `try` pentru a fi ținut sub observație. Chiar dacă utilizarea excepțiilor vă poate ajuta să faceți codul mai lizibil, prin separarea codului de tratare a erorilor de cel normal, totuși, utilizarea inadecvată a lor produce un cod dificil de citit și înțeles. Pe de altă parte, mecanismul Java de tratare a erorilor permite propagarea erorii, prin aruncarea mai departe a erorilor prinse, obligând altă metodă să le trateze. Nu în ultimul rând, verificarea de către compilator a existenței codului de tratare a unei excepții tratate, este un prim pas spre a avea un cod stabil și robust.

Din aceste motive este indicat ca pentru fiecare aplicație să se dezvolte o strategie de tratare a excepțiilor. De asemenea, excepțiile nu trebuie să fie considerate ca un adaos la o aplicație, ci ca o parte integrantă a acesteia, iar pentru a realiza acest lucru este necesar ca strategia de tratare a excepțiilor să fie adoptată în timpul fazei de proiectare a aplicației, și nu în faza de implementare.

Rezumat

Capitolul de față a prezentat modalitatea în care Java tratează situațiile de excepție în cadrul unei aplicații.

Excepțiile sunt folosite pentru a semnaliza situații de eroare. O excepție este generată de clauza `throw` și se propagă până este prinsă și tratată de un bloc `catch`, asociat cu un bloc `try`. Pe de altă parte, metodele specifică în cadrul unei liste `throws` excepțiile tratate pe care le aruncă.

Capitolul următor prezintă pe scurt sistemul de intrare/ieșire în limbajul Java.

Noțiuni fundamentale

`catch`: bloc de instrucțiuni utilizat pentru tratarea unei excepții.

excepție runtime: excepție netratată, care nu necesită să fie prinsă și tratată (ex. `NullPointerException`, `IndexOutOfBoundsException`).

excepție tratată: excepție care trebuie prinsă sau propagată explicit într-o clauză `throws`.

finally: bloc de instrucțiuni care se execută indiferent de rezultatul execuției blocului `try-catch`.

`NullPointerException`: tip de excepție generată de încercarea de a apela un atribut sau o metodă pe o referință `null`.

`throw`: instrucțiune utilizată pentru a arunca o excepție.

`throws`: indică faptul că o metodă poate propaga o excepție.

`try`: cuprinde codul suspect, care ar putea genera o excepție.

Erori frecvente

1. Excepțiile tratate trebuie să fie în mod obligatoriu prinse sau propagate în mod explicit prin intermediul clauzei `throws`.
2. Deși este tentant (de dragul simplității) să scriem clauze `catch` vide, aceasta face ca programul să fie foarte dificil de depanat, deoarece excepțiile sunt prinse fără a se semnala deloc acest lucru.

Exerciții

Pe scurt

1. Descrieți mecanismul de funcționare a excepțiilor în limbajul Java.
2. Care excepții trebuie tratate obligatoriu și care nu?

În practică

1. Creați o metodă `main()` care aruncă un obiect de tip `Exception` în interiorul unui bloc `try`. Furnizați constructorului obiectului `Exception` un mesaj de tip `String`. Prindeți excepția în cadrul unei clauze `catch` și afișați mesajul de tip `String`. Adăugați o clauză `finally` în care să tipăriți un mesaj pentru a verifica faptul că ați trecut pe acolo.
2. Creați o clasă proprie de excepții, asemănătoare lui `TeaTooHotException` din paragraful 6.3. Adăugați clasei un constructor cu un parametru de tip `String` care reține parametrul într-un atribut al clasei. Scrieți

- o metodă care afișează `String`-ul stocat. Exersați noua clasă creată folosind un bloc `try-catch`.
3. Scrieți o clasă cu o metodă care să arunce o excepție de tipul celei descrise la exercițiul anterior. Încercați să o compilați fără a preciza excepția în clauza `throws`. Adăugați apoi clauza `throws` adecvată și apelați metoda în cadrul unei clauze `try-catch`.
 4. Declarați o referință și inițializați-o cu `null`. Apelați apoi o metodă pe referința respectivă, pentru a verifica faptul că veți obține `NullPointerException`. Încadrați apoi apelul într-un bloc `try-catch` pentru a prinde excepția.
 5. Scrieți o secvență de cod care să genereze și să prindă o excepție de tipul `ArrayIndexOutOfBoundsException` sau de tipul `StringIndexOutOfBoundsException`.
 6. Definiți o clasă cu două metode, `f()` și `g()`. În metoda `f()` aruncați o excepție de un tip definit de dumneavoastră. În metoda `g()` apelați metoda `f()`, prindeți excepția aruncată de `f()` și aruncați mai departe o excepție de tip diferit. Testați codul în metoda `main()` a clasei.
 7. Definiți o metodă care aruncă (în clauza `throws`) mai multe tipuri de excepții definite de dumneavoastră. Apelați apoi metoda și prindeți toate excepțiile aruncate folosind o singură clauză `catch`. Modificați apoi programul astfel încât clauza `catch` să prindă doar aceste tipuri de excepții.
 8. Pentru exemplul cu clasa `Shape` din capitolul anterior, modificați metodele `readShape()` și `main()` astfel încât ele să arunce și să prindă excepții (în loc să creeze un cerc de rază 0) atunci când se observă o eroare la citire.
 9. În multe situații o metodă care aruncă o excepție este reapelată până când execuția ei decurge normal sau s-a atins un număr maxim de încercări. Creați un exemplu care să simuleze acest comportament, prin apelarea unei metode în cadrul unui ciclu `while`, care ciclează până când nu se mai aruncă nici o excepție.

7. Intrare și ieșire (Java I/O)

“Poți să îmi spui, te rog, pe unde să ies de aici?”

“Aceasta depinde în mare măsură de unde vrei să mergi”, spuse pisica.

“Nu prea îmi pasă unde...”, spuse Alice.

“Atunci nu contează pe ce drum ieși”, spuse pisica.

Lewis Carroll, Alice în țara minunilor

În capitolul anterior am prezentat excepțiile Java și modalitatea în care se tratează situațiile de eroare apărute în cadrul unei aplicații. În capitolul de față vom parcurge pe scurt sistemul Java de intrare-ieșire.

Vom afla detalii despre:

- Cum este structurat sistemul Java de intrare și ieșire (I/O);
- Care sunt modalitățile de a citi/scrie date într-o aplicație Java;
- Cum sunt utilizate pachetele de resurse pentru a simplifica citirea datelor de intrare într-un program Java.

7.1 Preliminarii

Intrarea și ieșirea în Java se realizează cu ajutorul claselor din pachetul predefinit `java.io`. Din acest motiv, orice program care folosește rutinele de intrare/ieșire trebuie să cuprindă instrucțiunea:

```
import java.io.*;
```

Biblioteca Java de intrare/ieșire este extrem de sofisticată și are un număr foarte mare de opțiuni. În cadrul acestui capitol vom examina doar elementele de bază referitoare la intrare și ieșire, concentrându-ne în întregime asupra operațiilor de intrare-ieșire formate (nu vom vorbi despre operații de intrare-ieșire binare).

Deși în practică programele Java își preiau datele de intrare din fișiere sau chiar prin Internet, pentru cei care învață acest limbaj este uneori mai convenabil să preia datele de la tastatură, așa cum obișnuiau în C sau Pascal. Vom arăta în paragrafele care urmează cum se pot prelua datele de la tastatură și cât de puține modificări sunt necesare pentru a prelua aceleași date din fișiere.

7.2 Operații de bază pe fluxuri (stream-uri)

Java oferă trei fluxuri predefinite pentru operații I/O standard:

- `System.in` (intrarea standard);
- `System.out` (ieșirea standard);
- `System.err` (fluxul de erori).

Așa cum am arătat deja, metodele `print` și `println` sunt folosite pentru afișarea formatată. Orice fel de obiect poate fi convertit la o formă tipăribilă, folosind metoda `toString()`; în multe cazuri, acest lucru este realizat automat. Spre deosebire de C și C++ care dispun de un număr enorm de opțiuni de formatare, afișarea în Java se face exclusiv prin concatenare de `String`-uri, fără nici o formatare.

O modalitate simplă de a realiza citirea este aceea de a citi o singură linie într-un obiect de tip `String` folosind metoda `readLine()`. `readLine()` preia caractere de la intrare până când întâlnește un terminator de linie sau sfârșit de fișier. Metoda returnează caracterele citite (din care extrage terminatorul de linie) ca un `String` nou construit. Pentru a citi date de la tastatură folosind `readLine()` trebuie să construim un obiect `BufferedReader` dintr-un obiect `InputStreamReader` care la rândul său este construit din `System.in`. Acest lucru a fost ilustrat în capitolul 3, **Listing 3.2**.

Dacă primul caracter citit este EOF, atunci metoda `readLine()` returnează `null`, iar dacă apare o eroare la citire, se generează o excepție de tipul `IOException`. În cazul în care șirul citit nu poate fi convertit la o valoare întreagă, se generează o excepție `NumberFormatException`.

7.3 Obiectul StringTokenizer

Obiectul `StringTokenizer` ne permite să separăm elementele lexicale distincte din cadrul unui string. Să ne reamintim că pentru a citi o valoare primitivă, cum ar fi `int`, foloseam `readLine()` pentru a prelua linia ca pe un `String` și apoi aplicam metoda `parseInt()` pentru a converti stringul la tipul primitiv.

În multe situații avem mai multe elemente pe aceeași linie. Să presupunem, de exemplu, că fiecare linie are două valori întregi. Putem accesa fiecare dintre aceste două valori, utilizând clasa `StringTokenizer` pentru a separa `String`-ul în elemente lexicale (engl. token). Pentru a putea folosi obiecte ale clasei `StringTokenizer` se folosește directiva:

```
import java.util.StringTokenizer;
```

Folosirea obiectului `StringTokenizer` este exemplificată în programul din **Listing 7.1**. Obiectul `StringTokenizer` este construit în linia 19 prin furnizarea obiectului `String` reprezentând linia citită. Apelul metodei `countTokens()` din linia 20 ne va da numărul de cuvinte citite (elemente lexicale). Metoda `nextToken()` întoarce următorul cuvânt necitit ca pe un `String`. Această ultimă metodă generează o excepție `NoSuchElementException` dacă nu există nici un cuvânt rămas, dar aceasta este o excepție de execuție standard și nu trebuie neapărat prinsă. La liniile 26 și 27 folosim `nextToken()` urmată de `parseInt()` pentru a obține un `int`. Toate erorile sunt prinse în blocul `catch`.

Implicit, Java consideră spațiul, caracterul *tab* sau linia nouă drept delimitator al elementelor lexicale, dar obiectele de tip `StringTokenizer` pot fi construite și în așa fel încât să recunoască și alte caractere drept delimitatori.

Listing 7.1: Program demonstrativ pentru clasa `StringTokenizer`

```
1 //program pentru exemplificarea clasei
2 //StringTokenizer. Programul citește două numere aflate pe
3 //aceeași linie și calculează maximul lor
4
5 import java.io.*;
6 import java.util.*;
7
8 public class TokenizerTest
9 {
10     public static void main(String[] args)
11     {
12         BufferedReader in = new BufferedReader(
13             new InputStreamReader(System.in));
14     }
```

```

15     System.out.println("Introduceți 2 nr. pe linie:");
16     try
17     {
18         String s = in.readLine();
19         StringTokenizer st = new StringTokenizer(s);
20         if (st.countTokens() != 2)
21         {
22             System.out.println("Numar invalid de " +
23                 "argumente!");
24             return;
25         }
26         int x = Integer.parseInt(st.nextToken());
27         int y = Integer.parseInt(st.nextToken());
28         System.out.println("Maximul este: " +
29             Math.max(x, y));
30     }
31     catch (Exception e)
32     {
33         System.out.println(e);
34     }
35 }
36 }

```

7.4 Fișiere secvențiale

Una dintre regulile fundamentale în Java este aceea că ceea ce se poate face pentru operații I/O de la terminal se poate face și pentru operații I/O din fișiere. Pentru a lucra cu fișiere, obiectul `BufferedReader` nu se construiește pe baza unui `InputStreamReader`. Vom folosi în schimb un obiect de tip `FileReader` care va fi construit pe baza unui nume de fișier.

Un exemplu pentru ilustrarea acestor idei este prezentat în **Listing 7.2**. Acest program listează conținutul fișierelor text care sunt precizate ca parametri în linie de comandă. Funcția `main()` parcurge pur și simplu argumentele din linia de comandă, apelând metoda `listFile()` pentru fiecare argument. În metoda `listFile()` construim obiectul de tip `FileReader` la linia 26 și apoi îl folosim pe acesta pentru construirea obiectului de tip `BufferedReader` în linia următoare. Din acest moment, citirea este identică cu cea de la tastatură.

După ce am încheiat citirea din fișier, trebuie să închidem fișierul, altfel s-ar putea să nu mai putem deschide alte stream-uri (să atingem limita maximă de stream-uri care pot fi deschise). Nu este indicat să facem acest lucru în primul bloc `try`, deoarece o excepție ar putea genera părăsirea prematură a blocului și fișierul nu ar fi închis. Din acest motiv, fișierul este închis după secvența `try/catch`.

Scrierea formatată în fișiere este similară cu citirea formatată.

Listing 7.2: Program care listează conținutul unui fișier

```

1 //program pentru afisarea continutului fisierelor
2 //ale caror nume sunt precizate in linia de comanda
3 import java.io.* ;
4
5 public class Lister
6 {
7     public static void main(String [] args)
8     {
9         if (args.length == 0)
10        {
11            System.out.println("Nici un fisier precizat!");
12        }
13
14        for (int i = 0 ; i < args.length; ++i)
15        {
16            listFile(args[i]);
17        }
18    }
19
20    public static void listFile(String fileName)
21    {
22        System.out.println("NUME FISIER: " + fileName);
23
24        try
25        {
26            FileReader file = new FileReader(fileName);
27            BufferedReader in = new BufferedReader(file);
28            String s;
29            while ((s = in.readLine()) != null)
30            {
31                System.out.println(s);
32            }
33        }
34        catch (Exception e)
35        {
36            System.out.println(e);
37        }
38        try
39        {
40            if (in != null)
41            {
42                in.close();
43            }
44        }
45        catch (IOException e)
46        { //ignora exceptia
47        }

```

```

48     }
49 }

```

7.5 Utilizarea colecțiilor de resurse (resource bundles)

Vom prezenta acum și o altă modalitate de a stoca și accesa date, în care se folosește o structurare a datelor la un nivel superior: colecțiile de resurse (resource bundles).

Scopul pentru care colecțiile de resurse au fost implementate de creatorii limbajului Java, nu a fost acela de a le folosi pentru citirea de date, ci pentru a realiza internaționalizarea aplicațiilor, cu alte cuvinte pentru a permite traducerea interfeței unei aplicații dintr-o limbă în alta, într-un mod foarte simplu. Utilizarea lor a fost însă extinsă de programatori și pentru a stoca și citi date într-un format mai organizat, superior orientării pe obiecte a fluxurilor de date. Vom prezenta în continuare un exemplu simplu de utilizare a colecțiilor de resurse pentru a extrage informații dintr-un fișier de proprietăți. Anexa D prezintă în detaliu modul în care se utilizează colecțiile de resurse și în alte direcții.

7.5.1 Utilizarea fișierelor de proprietăți pentru citirea datelor

Colecțiile de resurse pot fi reprezentate sub mai multe forme în Java. Anexa D vă stă la dispoziție pentru lămuriri în această privință. Ca exemplu, am ales pentru reprezentarea colecțiilor de resurse varianta *fișierelor de proprietăți*. Fișierele de proprietăți sunt simple fișiere text, cu extensia `.properties`. Iată un exemplu:

```

#fișierul de proprietati pentru programul Human.java
age=14
height=1.70

```

După cum se poate observa, fișierul de proprietăți este format din perechi de tipul cheie/valoare. Cheia este de partea stângă a semnului egal, iar valoarea este de partea dreaptă. De exemplu, `age` este cheia care corespunde valorii `14`. Analog, putem spune că `14` este valoarea care corespunde cheii `age`. Cheia este arbitrară. Am fi putut denumi cheia `age` altfel, de exemplu `myAge` sau `a`. Odată definită însă, cheia nu trebuie schimbată pentru că ea este referită în codul sursă. Pe de altă parte, valoarea asociată cheii poate fi schimbată fără probleme.

De asemenea, se remarcă faptul că semnul # este folosit pentru comentarea unei linii din cadrul fișierului de proprietăți.

Să considerăm că fișierul de proprietăți prezentat mai sus este salvat pe disc în fișierul cu numele `humanProp.properties`. Evident, puteți schimba acest nume cu unul la alegerea dumneavoastră. Programul Java din **Listing 7.3** prezintă o modalitate de utilizare a acestui pachet de resurse (pentru o execuție corectă, salvați acest fișier cu numele `Human.java` în același director cu fișierul `humanProp.properties`).

Listing 7.3: Program de test pentru fișiere de proprietăți

```

1 import java.util.*;
2
3 public class Human
4 {
5     public static void main(String args[])
6     {
7         try
8         {
9             ResourceBundle rb = ResourceBundle.getBundle(
10                 "humanProp");
11
12             String s1 = rb.getString("age");
13             System.out.println(s1);
14
15             String s2 = rb.getString("height");
16             System.out.println(s2);
17         }
18         catch(Exception e)
19         {
20             System.out.println("EXCEPTION: " + e.getMessage());
21         }
22     }
23 }

```

Utilizarea fișierelor de proprietăți este, după cum se vede și în exemplul precedent, destul de simplă. Metoda `getBundle()` realizează accesul la fișierul `humanProp.properties`. Se observă faptul că nu este necesară extensia `.properties` în apelul metodei.

Accesul la valorile cheilor se face prin metoda `getString()`. Trebuie reținut faptul că toate valorile care sunt preluate din fișierele `properties` sunt de tipul `String`. De aceea, ele trebuie convertite către tipurile lor. În cazul nostru, valoarea cheii `age` trebuie convertită la `int`, iar cea a cheii `height` la `float` sau `double`. Pentru a păstra simplitatea programului, am ales să nu mai convertim aceste valori, ci doar le afișăm pentru a testa valoarea lor. Rezultatul afișării este:

14

1.70

Această modalitate de organizare și citire a datelor este deosebit de utilă pentru a păstra date a căror organizare logică este de forma cheie=valoare, cum ar fi de exemplu stocarea opțiunilor utilizatorului unei aplicații. În cazul unui editor de texte scris în Java, fișierul de proprietăți ar putea stoca preferințele utilizatorului referitoare la dimensiunea fontului, culori, directorul de pornire etc.

Rezumat

Ceea ce am prezentat în acest capitol a constituit o prezentare sumară, dar suficientă pentru a putea merge mai departe, a sistemului de intrare-ieșire existent în Java. Au fost prezentate fluxurile de intrare/ieșire, clasa `StringTokenizer`, fișierele secvențiale și colecțiile de resurse. Deși introduse pentru a permite internaționalizarea aplicațiilor (vezi anexa dedicată acestei probleme), am văzut cum pot fi utilizate colecțiile de resurse pentru a prelua date din fișiere de proprietăți.

Noțiuni fundamentale

argument în linie de comandă: parametru scris în linia de comandă în care se realizează execuția programului respectiv. Acest parametru poate fi folosit în cadrul funcției `main` a aplicației rulate.

`BufferedReader`: clasă Java predefinită utilizată pentru citirea pe linii separate a datelor de intrare.

colecție de resurse: mulțime de proprietăți care au asociate anumite valori. Cunoscută și sub numele de pachet de resurse.

`FileReader`: clasă utilizată pentru lucrul cu fișiere.

fișiere de proprietăți: fișiere text având extensia `.properties`, cu un conținut special, format din perechi de tipul `proprietate=valoare`, stocate câte una pe linie.

`StringTokenizer`: clasă utilizată pentru a delimita cuvintele existente în cadrul unei propoziții. Cuvintele sunt despărțite prin separatori.

`System.out`, `System.in`, `System.err`: fluxurile standard de ieșire, intrare, eroare pentru operații de citire-scriere.

214

Erori frecvente

1. Fișierele de proprietăți trebuie să aibă extensia `.properties`.

Exerciții

1. Scrieți un program care afișează numărul de caractere, cuvinte, linii pentru fișiere precizate în linie de comandă.
2. Implementați un program care realizează copierea unui fișier dintr-o locație în alta.

8. Fire de execuție

Timpul este modalitatea prin care natura are grijă să nu se producă totul simultan.

Autor anonim

Programarea concurentă reprezintă pentru mulți programatori Java un concept neobișnuit, care necesită timp pentru a fi asimilat. Efortul necesar pentru tranziția de la programarea neconcurentă la programarea concurentă este asemănător cu cel pentru tranziția de la programarea procedurală la programarea orientată pe obiecte: o tranziție dificilă, frustrantă uneori, dar, în final, efortul este recompensat pe deplin. Dacă la prima parcurgere nu înțelegeți anumite secvențe din acest capitol, pentru a vă ușura sufletul dați vina pe slaba inspirație a autorilor, lăsați ceva timp pentru ca informațiile să se asimileze și încercați să rulați exemplele, după care reluați secvența care v-a creat dificultăți.

În acest capitol vom prezenta:

- Ce este un fir de execuție;
- Cum se creează aplicații care rulează pe mai mult fire de execuție;
- Cum se evită inconsistența la concurență folosind metode sincronizate și instrucțiunea `synchronized`;
- Cum funcționează monitoarele în Java;
- Cum se coordonează firele de execuție folosind metodele `wait()` și `notify()`.

8.1 Ce este un fir de execuție?

Firele de execuție (*engl.* threads) permit unui program să execute mai multe părți din el însuși în același timp. De exemplu, o parte a programului poate să realizeze o conexiune la un server de internet și să descarce niște date, în timp ce cealaltă parte a programului afișează fereastra principală și meniul aplicației pentru a putea interacționa în acest timp cu utilizatorul. Un alt exemplu clasic de utilizare a firelor de execuție îl constituie un browser de Web care vă permite să descărcați un număr nedefinit de fișiere (fiecare operație de descărcare reprezintă un fir de execuție), timp în care puteți să navigați nestingheriți prin alte pagini (pe un alt fir de execuție).

Firele de execuție sunt o invenție relativ recentă în lumea informaticii, deși *procesele*, surorile lor mai mari, sunt folosite deja de câteva decenii. Firele de execuție sunt asemănătoare cu procesele, în sensul că pot fi executate independent și simultan, dar diferă prin faptul că nu implică un consum de resurse atât de mare. Cei care ați programat în C sau C++ vă amintiți probabil faptul că prin comanda *fork* se putea crea un nou proces. Un proces astfel creat este o copie *exactă* a procesului original, având aceleași variabile, cod etc. Copierea tuturor acestor date face ca această comandă să fie costisitoare ca timp de execuție și resurse folosite. În timp ce rulează, procesele generate prin *fork* sunt complet independente de procesul care le-a creat, ele putându-și chiar modifica variabilele fără a afecta procesul părinte.

Spre deosebire de procese, firele de execuție nu fac o copie a întregului proces părinte, ci pur și simplu rulează o altă secvență de cod în același timp. Aceasta înseamnă că firele de execuție pot fi pornite rapid, deoarece ele nu necesită copierea de date de la procesul părinte. În consecință, firele de execuție vor partaja între ele datele procesului căruia îi aparțin. Ele vor putea scrie sau citi variabile pe care oricare alt fir de execuție le poate accesa. Din acest motiv, comunicarea între firele de execuție este mai facilă, dar poate conduce și la situația în care mai multe fire de execuție modifică aceleași date în același timp, conferind rezultate imprevizibile.

Firele de execuție permit o mai bună utilizare a resurselor calculatorului. În situația în care mașina virtuală Java (JVM) rulează pe un sistem multiprocesor, firele de execuție din cadrul aplicației pot să ruleze pe procesoare diferite, realizând astfel un paralelism real. Am arătat deja că există avantaje efective ale utilizării firelor de execuție și pe sistemele cu un singur procesor. Folosirea lor permite scrierea de programe mai simple și mai ușor de înțeles. Un fir de execuție poate să execute o secvență de cod care realizează o operație simplă, lăsând restul muncii altor fire de execuție.

În rezumat, putem spune că un *proces* este un program de sine stătător, care

dispune de propriul lui spațiu de adrese. Un sistem de operare multitasking este capabil să ruleze mai multe procese (programe) în același timp, oferindu-le periodic un anumit număr de cicluri procesor, lăsând astfel impresia că fiecare proces se execută de unul singur. Un fir de execuție (thread) reprezintă un singur flux de execuție din cadrul procesului. Astfel, un singur proces poate să conțină mai multe fire de execuție care se execută în mod concurrent.

8.2 Fire de execuție în Java

Specificările limbajului Java definesc firele de execuție (thread-urile) ca făcând parte din bibliotecile Java standard (pachetele `java.*`). Fiecare implementare a limbajului Java trebuie să furnizeze bibliotecile standard, deci implicit să suporte și fire de execuție. În consecință, cei care dezvoltă aplicații Java pot întotdeauna să se bazeze pe faptul că firele de execuție vor fi prezente pe orice platformă va rula programul. Mai mult, toate bibliotecile standard Java fie sunt *thread-safe* (adică sunt fiabile în cazul utilizării lor simultane de către mai multe fire de execuție) sau, dacă nu, au măcar un comportament bine definit în această situație. De exemplu, se știe că clasa `Vector` este thread-safe, deci elementele dintr-un vector pot fi modificate de către mai multe fire de execuție simultan, fără a se pierde consistența datelor. Clasa `ArrayList` este echivalentul clasei `Vector` care nu este thread-safe, dar are avantajul de a fi mai rapidă.

8.2.1 Crearea firelor de execuție

În Java există două variante pentru a crea fire de execuție: fie se derivează din clasa `Thread`, fie se implementează interfața `Runnable`. Ambele variante conduc la același rezultat. Dacă se implementează interfața `Runnable`, putem converti clase deja existente la fire de execuție fără a fi nevoie să modificăm clasele din care acestea sunt derivate (acesta este practic singurul avantaj adus de implementarea interfeței `Runnable`).

Derivarea din clasa `Thread`

Cea mai simplă modalitate de a crea un fir de execuție este să extindem clasa `Thread`, care conține tot codul necesar pentru a crea și rula fire de execuție. Cea mai importantă metodă pentru clasa `Thread` este `run()`. Prin redefinirea acestei metode se precizează instrucțiunile care trebuie executate de firul de

Listing 8.1: Exemplu simplu de fire de execuție create prin derivarea din clasa Thread

```

1 public class SimpleThread extends Thread
2 {
3     private int countDown = 5;
4     private static int threadCount = 0;
5     private int threadNumber ;
6
7     public SimpleThread()
8     {
9         threadNumber = ++threadCount ;
10        System.out.println("Creez firul nr. " + threadNumber);
11    }
12
13    public void run()
14    {
15        while(true)
16        {
17            System.out.println("Fir nr. " + threadNumber + "(" +
18                               countDown + ")");
19            if(--countDown == 0)
20            {
21                return;
22            }
23        }
24    }
25
26    public static void main(String[] args)
27    {
28        for(int i = 0; i < 3; i++)
29        {
30            new SimpleThread().start();
31        }
32        System.out.println("Firele de executie au fost pornite.");
33    }
34 }

```

execuție. Practic, metoda `run()` furnizează codul care va fi executat în paralel cu codul celorlalte fire de execuție din cadrul programului.

Programul din **Listing 8.1** creează 3 fire de execuție, fiecare fir având asociat un număr unic, generat cu ajutorul variabilei statice `threadCount`. Metoda `run()` este redefinită în liniile 13-24 pentru a scădea valoarea unui contor (variabila `countDown`) la fiecare execuție a ciclului `while` până când valoarea contorului ajunge la 0, moment în care metoda `run()` se încheie, iar firul de execuție este terminat.

În marea majoritate a cazurilor, instrucțiunile din metoda `run()` sunt cuprinse într-o instrucțiune de ciclare care se execută până când firul de execuție nu mai este necesar (în cazul nostru, până când contorul ajunge la 0).

În metoda `main()`, liniile 26-32, se crează și se pornesc 3 fire de execuție. Metoda `start()` din clasa `Thread` inițializează firul de execuție după care îi apelează metoda `run()`. Așadar, pașii pentru crearea unui fir de execuție sunt: se apelează constructorul pentru a crea obiectul, apoi metoda `start()` configurează firul de execuție și apelează metoda `run()`. Dacă nu se apelează metoda `start()` (lucru care poate fi făcut chiar și în constructor) firul de execuție nu va fi pornit niciodată.

Ieșirea afișată de programul din **Listing 8.1** (care *diferă* de la o execuție la alta) este:

```
Creez firul nr. 1
Creez firul nr. 2
Fir de executie nr. 1(5)
Fir de executie nr. 1(4)
Fir de executie nr. 1(3)
Fir de executie nr. 1(2)
Creez firul nr. 3
Fir de executie nr. 1(1)
Firele de executie au fost pornite.
Fir de executie nr. 3(5)
Fir de executie nr. 3(4)
Fir de executie nr. 3(3)
Fir de executie nr. 2(5)
Fir de executie nr. 3(2)
Fir de executie nr. 3(1)
Fir de executie nr. 2(4)
Fir de executie nr. 2(3)
Fir de executie nr. 2(2)
Fir de executie nr. 2(1)
```

Observați că firele de execuție nu au fost executate în ordinea în care au fost create. De fapt, ordinea în care procesorul rulează firele de execuție este nedeterminată, cu excepția situației în care se folosește metoda `setPriority()` pentru a modifica prioritatea de execuție a unui thread.

Un alt lucru demn de remarcat este că la crearea unui fir de execuție în linia 30, metoda `main()` nu reține nici o referință către obiectele nou create. Un obiect Java obișnuit ar fi în această situație disponibil pentru colectorul de

gunoaie (care, dacă tot veni vorba, rulează și el pe un fir de execuție separat), dar nu și un fir de execuție. Fiecare fir de execuție se înregistrează pe el însuși în cadrul mașinii virtuale, deci există undeva o referință către el, astfel încât colecatorul de gunoaie nu va putea să îl distrugă atât timp cât firul este în execuție.

Implementarea interfeței Runnable

Ajunși cu lectura în acest punct al cărții ar trebui să fiți familiarizați cu noțiunea de interfață, prezentată în capitolul 5. Implementarea interfeței Runnable reprezintă o modalitate elegantă prin care putem face ca orice clasă care o implementează să ruleze pe un fir de execuție distinct. Iată codul pentru interfața Runnable:

```
1 public interface Runnable extends Object
2 {
3     public abstract void run() ;
4 }
```

Așadar interfața Runnable este foarte simplă, însă deosebit de puternică. Toate clasele care implementează această interfață trebuie, desigur, să implementeze metoda `run()`. Este interesant de remarcat faptul că noul fir de execuție se va crea tot cu ajutorul clasei `Thread`, care oferă un constructor având ca parametru o instanță `Runnable`. Clasa `Thread` va executa în această situație metoda `run()` a parametrului în cadrul metodei sale `main()`. Firul de execuție se va încheia în momentul în care metoda `run()` se încheie. Astfel, această interfață foarte simplă permite ca oricare clasă care o implementează să ruleze pe un fir de execuție separat.

Ajunși în acest punct, v-ați pus probabil întrebarea: de ce este necesar să avem și clasa `Thread` și interfața `Runnable`? Există situații în care este mai adecvat să se implementeze interfața `Runnable` și situații în care este de preferat să se extindă clasa `Thread`. Crearea unui fir de execuție prin implementarea interfeței `Runnable` este mai flexibilă, deoarece poate fi întotdeauna utilizată. Nu veți putea întotdeauna să extindeți clasa `Thread`. Adevărata putere a interfeței `Runnable` iese în evidență în momentul în care aveți o clasă care trebuie să extindă o altă clasă pentru a moșteni un anumit gen de funcționalitate. În același timp, ați dori ca acea clasă să ruleze pe un fir de execuție separat. Aici este cazul să folosiți `Runnable`.

Odată ce aveți o clasă care implementează `Runnable` și definește o metodă `run()`, sunteți pregătiți pentru a crea un fir de execuție. Și în această situație va trebui să instanțiați clasa `Thread`. Fiecare fir de execuție în Java este asociat cu o instanță a clasei `Thread`, chiar dacă se utilizează interfața `Runnable`. Așa cum am amintit deja, clasa `Thread` oferă un constructor care primește

ca parametru o instanță `Runnable`. La instanțierea clasei `Thread`, cu acest constructor, metoda `run()` a clasei `Thread` apelează pur și simplu metoda `run()` a clasei primită ca parametru. Puteți verifica acest lucru cu ușurință, executând următorul program:

```

1 public class TestRunnable implements Runnable
2 {
3     public static void main(String[] args)
4     {
5         new Thread( new TestRunnable() ).start();
6     }
7     public void run()
8     {
9         new Exception().printStackTrace();
10        return ;
11    }
12 }

```

Programul de mai sus reprezintă un exemplu simplu de creare a unui fir de execuție prin implementarea interfeței `Runnable`. La linia 1 declarăm faptul că clasa noastră implementează interfața `Runnable`. În liniile 7-11 definim metoda `run()` care nu face decât să apeleze metoda `printStackTrace()` a clasei `Exception`, după care se încheie și, odată cu ea, se încheie și firul de execuție. La linia 5 se pornește firul de execuție, prin crearea unei instanțe a clasei `Thread` având ca parametru `Runnable` chiar clasa noastră. Rezultatul afișat de program este:

```

java.lang.Exception at
TestRunnable.run(TestRunnable.java:9) at
java.lang.Thread.run(Thread.java:484)

```

ceea ce dovedește că metoda `run()` a clasei `TestRunnable` a fost apelată de metoda `run()` a clasei `Thread`.

Programul din **Listing 8.2** reprezintă varianta `Runnable` a programului din **Listing 8.1**. Remarcați diferențele minime între cele două variante. La linia 1, clasa `SimpleRunnable` declară că implementează interfața `Runnable`. Apoi, la linia 30 firul de execuție se creează transmițând a instanță a clasei `SimpleRunnable` constructorului clasei `Thread`. În rest, cele două programe sunt identice.

8.3 Accesul concurent la resurse

Un program care rulează pe un singur fir de execuție poate fi privit ca o unică entitate care se mișcă prin spațiul problemei făcând un singur lucru la

Listing 8.2: Exemplu simplu de fire de execuție create prin implementarea interfeței Runnable

```

1 public class SimpleRunnable implements Runnable
2 {
3     private int countDown = 5;
4     private static int threadCount = 0;
5     private int threadNumber ;
6
7     public SimpleRunnable()
8     {
9         threadNumber = ++threadCount ;
10        System.out.println("Creez firul nr. " + threadNumber);
11    }
12
13    public void run()
14    {
15        while(true)
16        {
17            System.out.println("Fir nr. " + threadNumber + "(" +
18                               countDown + ")");
19            if(--countDown == 0)
20            {
21                return;
22            }
23        }
24    }
25
26    public static void main(String [] args)
27    {
28        for(int i = 0; i < 3; i++)
29        {
30            new Thread( new SimpleRunnable() ).start();
31        }
32        System.out.println("Firele de executie au fost pornite.");
33    }
34 }

```

un moment dat. Deoarece există doar o singură entitate nu este necesar să ne punem problema dacă două entități ar putea accesa aceeași resursă în același timp, cum ar fi de exemplu situația a două persoane care își parchează mașina în același loc simultan, sau care încearcă să iasă în același timp pe ușă, sau chiar să vorbească în același timp.

În cazul mai multor fire de execuție, nu mai avem o singură entitate și există deci posibilitatea ca două sau mai multe fire de execuție să încerce să utilizeze aceeași resursă în același timp. Nu întotdeauna accesarea unei resurse simultan este o problemă, dar sunt situații în care aceste coleziuni trebuie prevenite (de exemplu, două fire de execuție care accesează simultan același cont în bancă, sau care doresc să tipărească la aceeași imprimantă etc.).

Exemplul simplu de mai jos pune în evidență problemele care pot să apară într-un program multithreaded (care rulează pe mai multe fire de execuție):

```

1 public class Counter
2 {
3     private int count = 0 ; //counter incrementat de increment()
4     public int increment()
5     {
6         int aux = count ;
7         count++ ;
8         return aux ;
9     }
10 }

```

Clasa de mai sus este extrem de simplă, având un singur atribut și o singură metodă. Așa cum îi sugerează și numele, clasa Counter este utilizată pentru a număra diverse lucruri, cum ar fi numărul de apăsări ale unui buton, sau numărul de accesări al unei pagini web. Esența clasei este metoda `increment()` care incrementează și returnează valoarea curentă a counter-ului. Deși clasa Counter pare complet inofensivă, ea consituie o potențială sursă de comportament imprevizibil într-un mediu multithreaded.

Să considerăm ca exemplu un program Java care conține două fire de execuție, amândouă fiind pe cale să execute următoarea linie de cod:

```
int count = counter.increment() ;
```

Programatorul nu poate nici să controleze și nici să prezică ordinea în care cele două fire de execuție vor fi executate. Sistemul de operare (sau mașina virtuală Java) deține controlul complet asupra planificării firelor de execuție. În consecință, nu există nici o certitudine în privința cărui fir de execuție se va executa, sau cât timp va fi executat un anumit fir de execuție. Orice fir de execuție poate fi întrerupt printr-o schimbare de context în orice moment. Mai mult, două fire de execuție pot rula simultan pe două procesoare separate ale unei mașini multiprocesor.

Tabela 8.1 descrie o posibilă secvență de execuție a celor două fire de execuție. În acest scenariu, primul fir de execuție este lăsat să ruleze până când încheie apelul către metoda `increment()`; apoi, cel de-al doilea fir face același lucru. Nu este nimic surprinzător la acest scenariu. Primul fir va incrementa valoarea contorului la 1, iar al doilea o va incrementa la 2.

Tabela 8.1: Counter - Scenariul 1

Fir de execuție 1	Fir de execuție 2	count	aux
<code>count = counter.increment()</code>	—	0	-
<code>int aux = count //0</code>	—	0	0
<code>count++ //1</code>	—	1	0
<code>return aux //0</code>	—	1	0
—	<code>count = counter.increment()</code>	1	-
—	<code>aux = count //1</code>	1	1
—	<code>count++ //2</code>	2	1
—	<code>return aux //1</code>		1

Tabela 8.2 descrie o secvență de execuție oarecum diferită. În această situație, primul fir de execuție este întrerupt de o schimbare de context chiar în timpul execuției metodei `increment()`. Primul fir este astfel suspendat temporar și se permite execuția celui de-al doilea. Al doilea fir execută apelul către metoda `increment()`, măbind valoarea contorului la 1 și returnând valoarea 0. Când primul fir își va relua execuția, problema devine deja evidentă. Valoarea contorului este crescută la 2, dar valoarea returnată este tot 0!

Tabela 8.2: Counter - Scenariul 2

Fir de execuție 1	Fir de execuție 2	count	aux
<code>count = counter.increment()</code>	—	0	-
<code>int aux = count //0</code>	—	0	0
—	<code>count = counter.increment()</code>	0	-
—	<code>aux = count //0</code>	0	0
—	<code>count++ //1</code>	1	0
—	<code>return aux //0</code>	0	0
<code>count++ //2</code>	—	2	0
<code>return aux //0</code>	—	2	0

Examinând scenariul din **Tabela 8.2**, veți observa o secvență interesantă de

execuție a operațiilor. La intrarea în metoda `increment()`, valoarea atributului `count` (0) este stocată în variabila locală `aux`. Primul fir este apoi suspendat pentru o perioadă, timp în care se execută celălalt fir (este important să observăm că valoarea lui `count` se modifică în această perioadă). Când primul fir își reia execuția, el incrementează corect valoarea lui `count` și returnează valoarea din `aux`. Problema este că valoarea din `aux` nu mai este corectă, deoarece valoarea lui `count` a fost modificată între timp de către celălalt fir. Pentru a ne convinge de faptul că scenariul din **Tabela 8.2** este într-adevăr posibil, am extins clasa `Counter` prezentată anterior, adăugându-i clasa interioară (vezi paragraful 5.7, pagina 145) `CounterTest` (**Listing 8.3**) derivată din `Thread`. Metoda `run()` a clasei `CounterTest` (liniile 34-42) apelează într-un ciclu infinit metoda `increment()` a clasei `Counter`. Variabilele `check1` și `check2` declară și inițializează câte un obiect de tip `CounterTest` care rulează pe un fir de execuție separat (firul de execuție este pornit chiar în constructorul clasei `CounterTest` care apelează metoda `start()` în linia 30).

Listing 8.3: Clasă Java care evidențiază problemele care pot apărea la partajarea resurselor

```

1 public class Counter
2 {
3     private int count = 0 ; //counter incrementat de increment()
4     //initializeaza si porneste primul thread de verificare
5     private CounterTest check1 = new CounterTest() ;
6     //initializeaza si porneste al doilea thread de verificare
7     private CounterTest check2 = new CounterTest() ;
8
9     public int increment()
10    {
11        int aux = count ;
12        count++ ;
13        if ( count != aux+1 )
14        {
15            System.out.println("Oops! Ceva nu e in regula ... " +
16                               this.count + " " + aux) ;
17        }
18        return aux ;
19    }
20
21    /**
22     * Clasa care evidentiaza problemele de sincronizare
23     * ale metodei increment.
24     */
25    public class CounterTest extends Thread
26    {
27        /** Construiește și porneste firul de execuție.*/

```

```

28  public CounterTest()
29  {
30      this.start();
31  }
32
33  /** Apeleaza metoda increment() intr-un ciclu infinit.*/
34  public void run()
35  {
36      System.out.println("Firul " + this.getName() +
37                          " a pornit");
38      for(;;)
39      {
40          increment();
41      }
42  }
43  }
44
45  public static void main( String[] args )
46  {
47      //creaza o instanta a clasei Counter
48      new Counter();
49  }
50  }

```

În mod aparent paradoxal, după secvența de instrucțiuni

```

int aux = count;
count++;

```

la linia 13 se verifică dacă valoarea variabilei `count` este diferită de `aux+1`

```

if( count != aux+1 )

```

Acest test ar fi complet inutil într-un program care rulează pe un singur fir de execuție, dar nu și în cazul nostru. La rularea programului `Counter.java`, acesta va afișa (valorile afișate diferă, desigur, de la o execuție la alta):

```

Firul Thread-0 a pornit
Firul Thread-1 a pornit
Oops! Ceva nu e in regula... Thread-1 18777999
18247501
Oops! Ceva nu e in regula... Thread-1 55129926
54599470
Oops! Ceva nu e in regula... Thread-0 81760783
81229276
Oops! Ceva nu e in regula... Thread-0 104069819
103556887
Oops! Ceva nu e in regula... Thread-0 118311248

```

```

117779680
Oops! Ceva nu e in regula... Thread-1 279005154
278475302

```

Din cele afișate reiese că testul de la linia 13 nu este deloc inutil. Se poate întâmpla ca unul dintre firele de execuție ale clasei `CounterTest` (să zicem `check1`) să fie întrerupt imediat după ce a fost atribuită valoarea lui `count` variabilei `aux`, dar înainte ca metoda să apuce să returneze valoarea din `aux` (ca în scenariul din **Tabela 8.2**). În această situație, celălalt fir de execuție al clasei `CounterTest`, `check2`, va continua să incrementeze valoarea lui `count` de un număr nedeterminat de ori, după care controlul va fi din nou preluat de firul de execuție întrerupt, `check1`. Acesta va avea noua valoare a lui `count`, care a fost între timp incrementată, dar vechea valoare a lui `aux`, care (fiind variabilă locală) a fost salvată pe stivă înainte de schimbarea de context¹, de aici rezultând inconsistența semnalată de testul din linia 13. Desigur, șansele ca primul fir de execuție să fie întrerupt chiar după ce a fost incrementat `count` sunt foarte mici, dovadă și faptul că această situație a apărut după peste 18 milioane de apeluri ale metodei `increment()`. Totuși, șansele există, și neluarea lor în considerare poate avea consecințe grave asupra funcționării unui program.

Problema pusă în evidență de scenariul din **Tabela 8.2** poartă numele de *inconsistență la concurență* (engl. *race condition*)- rezultatul unui program depinde de ordinea în care s-au executat firele de execuție pe procesor. În general, nu este deloc indicat să se permită incosistențe la concurență în rezultatele unui program. Noi nu putem ști niciodată în ce moment un fir de execuție este executat sau întrerupt. Închipuiți-vă că sunteți așezat la o masă cu o lingură în mână gata-gata să gustați dintr-o salată de fructe delicioasă care se află în fața dumneavoastră. Vă îndreptați plini de speranță lingura către salata de fructe și, chiar în momentul în care să atingeți salata, aceasta dispare fără urmă (deoarece firul vostru de execuție a fost suspendat, și între timp, alt fir de execuție a furat mâncarea). Mergând cu analogia mai departe, dumneavoastră nu aveți timp să reacționați la dispariția farfuriei (fiecare fir de execuție are iluzia că execută singur și continuu pe procesor) și veți înfige lingura în masă, provocând astfel îndoirea ei (adeseori, folosirea simultană a unei resurse conduce la coruperea ei).

Există situații în care nu are importanță dacă o resursă este accesată în același timp de mai multe fire de execuție (mâncarea este pe o altă farfurie). Totuși, pentru ca programele *multithreaded* (rom. cu mai multe fire de execuție)

¹Deși nu am precizat clar acest lucru, variabilele locale sunt distincte pentru fiecare fir de execuție în parte. Totuși, atributele claselor sunt partajate de către toate firele de execuție care rulează la un moment dat.

să funcționeze, trebuie să existe un mecanism care să permită împiedicarea accesului la o resursă de către mai multe thread-uri, măcar în timpul perioadelor critice. Să ne imaginăm ce s-ar fi întâmplat dacă metoda `increment()` ar fi fost utilizată pentru a actualiza contul în bancă al unei persoane în loc să incrementeze pe `count`. Pentru o bancă, corectitudinea conturilor este de importanță maximă. Dacă o bancă face erori de calcul sau raportează informații incorecte, cu siguranță nu își va încânta clienții.

Orice programe multithreaded, chiar și cele scrise în Java, pot suferi de inconsistență la concurență. Din fericire, Java ne pune la dispoziție un mecanism simplu pentru a controla concurența: *monitoarele*.

8.3.1 Sincronizare

Există multe lucrări de programare și sisteme de operare care se ocupă de problema programării concurente. Concurența a fost un subiect foarte cercetat în ultimii ani. Au fost propuse și implementate mai multe mecanisme de control al concurenței, printre care:

- secțiunile critice;
- semafoare;
- mutexuri;
- blocarea de înregistrări în baze de date;
- monitoare.

Limbajul Java implementează o variantă a monitoarelor (engl. *monitors*) pentru controlul concurenței. Conceptul de *monitor* a fost introdus de C.A.R. Hoare (același Hoare care a propus și algoritmul de partiționare de la Quicksort) în anul 1974 într-o lucrare publicată în *Communications of the ACM*. Hoare descrie în această lucrare un obiect special, denumit monitor, care este utilizat pentru a furniza excluderea reciprocă pentru un grup de proceduri (“excludere reciprocă” este un mod elegant de a spune “Un singur fir de execuție la un moment dat”). În modelul propus de Hoare, fiecare grup de proceduri care necesită excludere reciprocă este pus sub controlul unui monitor. În timpul execuției, monitorul permite să se execute doar o singură procedură aflată sub controlul lui la un moment dat. Dacă un alt fir de execuție încearcă să apeleze o procedură controlată de monitor, acel fir este suspendat până în momentul în care firul curent își încheie apelul.

Monitoarele din Java implementează îndeaproape conceptele inițiale ale lui Hoare, aducând doar câteva modificări minore (pe care nu le vom prezenta în această carte). Monitoarele din Java impun excluderea reciprocă la accesarea de metode, sau, mai exact, excluderea reciprocă la accesarea de metode sincronizate (vom vedea imediat un exemplu concret de metodă sincronizată).

În momentul în care se apelează o metodă Java sincronizată se declanșează un proces deosebit de laborios. În primul rând, mașina virtuală Java va localiza monitorul asociat cu obiectul a cărui metodă este apelată (de exemplu, dacă se apelează `counter.increment()`, mașina virtuală va localiza monitorul lui `counter`). Orice obiect Java poate să aibă un monitor asociat, deși, din motive de eficiență, mașina virtuală crează monitoarele doar atunci când este nevoie de ele. Odată ce monitorul este găsit, mașina virtuală încearcă să atribuie monitorul firului de execuție care a apelat metoda sincronizată. Dacă monitorul nu este atribuit altui fir de execuție, se va atribui firului curent, căruia îi este apoi permis să continue cu apelul metodei. În situația în care monitorul este deja deținut de către un alt fir de execuție, el nu va putea fi atribuit firului curent, care va fi pus în așteptare până când monitorul va fi eliberat. În momentul în care monitorul va deveni disponibil, aceasta va fi atribuit firului de execuție curent, care va putea continua cu apelul metodei.

Metaforic vorbind, un monitor acționează ca un fel de secretară aflată la intrarea în birou a unei persoane foarte solicitate. În momentul în care se apelează o metodă sincronizată, secretara permite firului apelant să treacă, după care închide ușa de la birou. Atâta timp cât firul este în cadrul metodei sincronizate, apelurile către alte metode sincronizate ale obiectului sunt blocate. Aceste fire se așează la coadă în fața ușii de la birou, așteptând cu răbdare ca primul fir să plece. În momentul în care primul fir părăsește metoda sincronizată, secretara deschide din nou ușa de la birou permițând unui singur fir din coada de așteptare să intre pentru a-și executa metoda sincronizată, după care procesul se repetă din nou.

Mai simplu spus, un monitor Java impune un acces de tipul un singur fir la un moment dat. Această operație mai este numită adeseori și serializare (a nu se confunda cu serializarea obiectelor, care permite să citim și să scriem obiecte Java într-un *stream* (rom. flux de date). Aici serializare are sensul de a permite accesul succesiv).

Observație: Programatorii care sunt deja familiarizați cu conceptul de programare multithreaded din alte limbaje de programare, tind adeseori să confunde monitoarele cu secțiunile critice. Declararea unei metode sincronizate nu implică faptul că doar un singur fir de execuție va executa acea metodă la un moment dat, cum este cazul la secțiunile critice. Ea impune doar ca numai

un singur fir de execuție să poată executa acea metodă (sau oricare altă metodă sincronizată) *pe un anumit obiect* la un moment dat. Monitoarele din Java sunt asociate cu obiecte, și nu cu linii de cod. Este foarte posibil ca două fire de execuție să execute simultan aceeași metodă sincronizată, cu condiția ca metoda să fie invocată pe obiecte distincte (adică `a.method()` și `b.method()`, unde `a != b`).

Pentru a vedea cum operează monitoare, să rescriem clasa `Counter`, astfel încât să se folosească de monitoare, prin intermediul cuvântului cheie `synchronized`.

Listing 8.4: Clasa `Counter` cu metoda sincronizată

```

1 public class CounterSync
2 {
3     private int count = 0 ; //counter incrementat de increment()
4     public synchronized int increment()
5     {
6         int aux = count ;
7         count++ ;
8         return aux ;
9     }
10 }
```

Observați că nu a fost necesar să rescriem metoda `increment()` - aceasta a rămas la fel ca în exemplul anterior, cu deosebirea că metoda a fost declarată a fi `synchronized`.

Ce s-ar întâmpla dacă am folosi clasa `CounterSync` în programul din **Listing 8.3**? Ar mai fi posibilă apariția scenariului din **Tabela 8.2**, care pune în evidență inconsistența la concurență? Rezultatul aceleiași secvențe de schimbări de context, nu ar mai fi identic, deoarece metoda sincronizată va împiedica inconsistența la concurență. Scenariul modificat din **Tabela 8.2** este prezentat în **Tabela 8.3**.

Tabela 8.3: Counter - Scenariul 3

Fir de execuție 1	Fir de execuție 2	count	aux
count = counter.increment()	—	0	-
(obține monitorul)	—	0	-
int aux = count //0	—	0	0
—	count = counter.increment()	0	-
—	(nu poate obține monitorul)	0	-
count++ //1	— blocat	1	0
return aux //0	— blocat	1	0
(eliberează monitorul)	— blocat	1	0
—	(obține monitorul)	1	-
—	int aux = count //1	1	1
—	count++ //2	2	1
—	return aux //1	2	1
—	(eliberează monitorul)		

Secvența de operații din **Tabela 8.3** începe la fel ca cea din scenariul anterior: primul fir de execuție începe să execute metoda `increment()` a clasei `CounterSync`, după care este întrerupt de o schimbare de context. Totuși, de data aceasta, în momentul în care al doilea fir de execuție dorește să apeleze metoda `increment()` pe același obiect de tip `CounterSync`, acesta este blocat. Tentativa celui de-al doilea fir de a obține monitorul obiectului eșuează, deoarece acesta este deja deținut de primul fir. În momentul în care primul fir eliberează monitorul, al doilea fir îl va putea obține și va executa metoda `increment()`. Putem să ne convingem de aceasta, modificând programul din **Listing 8.3** prin adăugarea cuvântului `synchronized` la metoda `increment()`. La execuție programul va afișa:

```
Firul Thread-0 a pornit
Firul Thread-1 a pornit
```

după care va rula până când va fi oprit cu CTRL-C, fără a mai afișa nici un mesaj.

Cuvântul cheie `synchronized` este singura soluție pe care limbajul Java o oferă pentru controlul concurenței. Așa cum s-a văzut și în exemplul cu clasa `Counter`, incosistența potențială la concurență a fost eliminată prin adăugarea modificatorului `synchronized` la metoda `increment()`. Astfel, toate apelurile metodei `increment()` au fost serializate. În general vorbind, modificatorul `synchronized` ar trebui adăugat la orice metodă care modifică

atributele unui obiect. Adeseori este deosebit de dificil să examinăm metodele unei clase pentru a detecta posibile inconsistențe la concurență. Este mult mai comod să marcăm toți modificatorii ca fiind `synchronized`, rezolvând astfel orice problemă de concurență.

Observație: Unii programatori mai curioși s-ar putea întreba când o să vadă și ei un monitor Java. Totuși, acest lucru nu este posibil. Monitoarele Java nu dispun de o descriere oficială în specificația Java, iar implementarea lor nu este direct vizibilă pentru un programator. Monitoarele nu sunt obiecte Java: ele nu au atribute sau metode. Monitoarele sunt un concept aflat la baza implementării modelului de concurență în Java. Se poate accesa monitorul unui obiect la nivel de cod nativ, dar acest lucru nu este recomandat, iar descrierea lui depășește cadrul acestui capitol.

8.3.2 Capcana metodelor nesincronizate

Monitoarele din Java sunt folosite doar în conjuncție cu metodele `synchronized`². Metodele care nu sunt declarate ca fiind `synchronized` nu încearcă să obțină monitorul obiectului înainte de a fi executate. Ele pur și simplu ignoră complet monitoarele. La un moment dat, cel mult un fir de execuție poate să execute o metodă sincronizată a unui obiect, dar un număr arbitrar de fire de execuție pot executa metode nesincronizate. Acest fapt poate să prindă pe picior greșit mulți programatori începători, care nu sunt suficient de atenți la care metode trebuie declarate `synchronized` și care nu. Să considerăm clasa `Account` din **Listing 8.5**, care descrie un cont în bancă împreună cu operațiile care se realizează asupra lui.

Listing 8.5: Clasa `Account`

```
1 public class Account
2 {
3     private int balance;
4     /** Creeaza un cont cu soldul balance */
5     public Account(int balance)
6     {
7         this.balance = balance;
8     }
9     /**
10      * Transfera suma amount din contul curent
11      * in contul destination
12      */
```

²Afirmația este un pic inexactă. Vom vedea imediat că monitoarele Java sunt folosite și în cazul instrucțiunii `synchronized()`.

```

13 public synchronized void transfer(int amount,
14                                   Account destination)
15 {
16     this.withdraw(amount);
17     destination.deposit(amount);
18 }
19 /** Retrage suma amount din cont */
20 public synchronized void withdraw(int amount)
21 {
22     balance -= amount;
23 }
24 /** Depune suma amount in cont */
25 public synchronized void deposit(int amount)
26 {
27     balance += amount;
28 }
29 /** Returneaza soldul din cont */
30 public int getBalance()
31 {
32     return balance;
33 }
34 }

```

Toți modificatorii clasei `Account` sunt declarați ca fiind `synchronized`. Aparent, această clasă nu ar trebui să aibă nici un fel de probleme legate de inconsistența la concurență. Și totuși, ea are!

Pentru a înțelege inconsistența la concurență de care suferă clasa `Account`, să vedem cum lucrează o bancă cu conturile. Pentru o bancă este de o importanță majoră să mențină corectitudinea datelor din conturi. Pentru a evita raportarea de informații inconsistente, băncile preferă ca, în timp ce se realizează o tranzacție, să dezactiveze cererile privitoare la soldul dintr-un cont. Astfel, clienții băncii nu pot să primească informații preluate în timpul unei tranzacții parțial încheiate. Metoda `getBalance()` din clasa `Account` nu este sincronizată, iar acest fapt poate conduce la anumite probleme.

Să presupunem că un fir de execuție apelează metoda sincronizată `transfer()`, pentru a trece suma de 10 milioane de lei din contul curent în alt cont. Mai mult, firul nostru de execuție este întrerupt de o schimbare de context imediat după ce a executat linia 16:

```
this.withdraw(amount);
```

În acest moment, suma de 10 milioane a fost extrasă din primul cont, dar încă nu a fost depozitată în celălalt cont. Dacă un alt fir de execuție va apela metoda (nesincronizată!) `getBalance()` pentru cele două conturi implicate în transfer, aceasta va putea accesa nestingherită informația din cele două conturi. Astfel, `getBalance()` va raporta că suma de 10 milioane de lei a fost

extrasă din primul cont, dar această sumă nu se va regăsi în contul destinație, deoarece metoda `transfer()` nu a apucat să actualizeze acest cont înainte de a fi întreruptă! Astfel, clienții s-ar putea întreba unde a dispărut suma respectivă³. Dacă `getBalance()` ar fi fost sincronizată, acest scenariu nu ar fi fost posibil, deoarece apelul ei ar fi fost blocat până la încheierea tranzacției.

8.3.3 Instrucțiunea `synchronized`

Metodele sincronizate nu pot fi folosite în orice situație. De exemplu, șirurile Java nu dispun de metode, cu atât mai puțin de metode sincronizate. Există totuși multe cazuri în care dorim ca accesul la elementele unui șir să fie sincronizat. Pentru a rezolva această situație, limbajul Java oferă și o altă convenție sintactică prin care putem interacționa cu monitorul unui obiect: instrucțiunea `synchronized`, cu următoarea sintaxă:

```
synchronized ( expresie )
    instructiune
urmatoarea instructiune
```

Execuția unei instrucțiuni `synchronized` are același efect ca și apelarea unei metode sincronizate: firul de execuție va trebui să dețină un monitor înainte de a executa instrucțiune (care poate fi simplă sau compusă). În cazul instrucțiunii `synchronized` se va utiliza monitorul obiectului obținut prin evaluarea lui `expresie` (al cărei rezultat nu trebuie să fie un tip primitiv).

Instrucțiunea `synchronized` este cel mai adesea folosită pentru a serializa accesul la obiecte de tip `array`. Secvența de cod de mai jos este un exemplu de cum se poate serializa accesul concurențial la elementele unui șir:

```
1 /** Incrementează elementul de pe poziția poz. */
2 void increment(byte[] sir, int poz)
3 {
4     synchronized ( sir )
5     {
6         sir[ poz ]++ ;
7     }
8 }
```

În exemplul anterior, mașina virtuală Java va atribui monitorul lui `sir`⁴ firului de execuție curent, înainte de a încerca să incrementeze elementul de pe poziția `poz`. Alte fire de execuție care ar încerca să obțină monitorul acestui șir

³Acest exemplu nu este concludent pentru multe bănci din România, care au nevoie de un interval de 2 zile pentru ca suma transferată dintr-un cont să apară în celălalt cont...

⁴Deși nu sunt obiecte în adevăratul sens al cuvântului, șirurile din Java au anumite proprietăți specifice obiectelor, printre care și monitorul.

vor fi forțate să aștepte până când se încheie operația de incrementare. Trebuie însă să avem în vedere faptul că accesul la elementele șirului va fi totuși permis în cadrul altor instrucțiuni care nu sunt sincronizate pe monitorul șirului.

Instrucțiunea `synchronized` este utilă și atunci când dorim să modificăm un obiect fără a folosi metode sincronizate. Această situație poate să apară atunci când se modifică atributele publice ale unui obiect (deși am arătat deja că utilizarea de atribute publice nu este de dorit) sau se apelează o metodă care nu este sincronizată (dar ar trebui să fie) și noi nu o putem modifica (de exemplu, pentru că este moștenită de la un alt obiect). Iată un exemplu:

```
1 void metoda( OClasa obiect )
2 {
3     synchronized( obiect )
4     {
5         obiect.metoda_care_ar_trebui_sa_fie_sincronizata_dar_nu_e();
6     }
7 }
```

O metodă sincronizată este echivalentă cu o metodă nesincronizată ale cărei instrucțiuni sunt cuprinse într-un bloc `synchronized` pe monitorul obiectului (`this`). Astfel, metoda

```
1 synchronized void f()
2 {
3     instructiune ;
4 }
```

este echivalentă cu metoda

```
1 void f()
2 {
3     synchronized( this )
4     {
5         instructiune ;
6     }
7 }
```

Observație: Instrucțiunea `synchronized` ne oferă posibilitatea de a utiliza monitoarele oricărui obiect Java. Totuși, folosirea instrucțiunii `synchronized` în situații în care ar fi fost suficientă declararea unei metode sincronizate duce uneori la un cod greoi și dificil de depanat. Adăugarea modificatorului `synchronized` la antetul unei metode definește foarte clar ce se petrece atunci când metoda este apelată.

8.3.4 Competiție asupra monitoarelor

Se poate întâmpla ca două sau mai multe fire de execuție să fie blocate în așteptarea obținerii aceluiași monitor. Această situație apare atunci când un fir de execuție deține monitorul unui anumit obiect. Dacă un alt fir de execuție încearcă să execute una din metodele sincronizate ale aceluiași obiect, firul va fi suspendat, în așteptarea eliberării monitorului. Dacă un al treilea fir de execuție încearcă și el să apeleze o metodă sincronizată a obiectului, va fi și el suspendat. În momentul în care monitorul va deveni disponibil, vor fi două fire de execuție care așteaptă să îl obțină.

În situația în care două sau mai multe fire de execuție așteaptă obținerea aceluiași monitor, mașina virtuală Java va trebui să aleagă doar un singur fir și să îi atribuie acestuia monitorul. Nu există nici o garanție în privința regulii după care mașina virtuală va decide care fir va fi ales. Specificația limbajului Java prevede ca doar un singur fir de execuție să obțină monitorul, fără a preciza un algoritm după care mașina virtuală să aleagă firul de execuție. De exemplu, mașina virtuală Java care rulează pe Solaris, alege firul de execuție în funcție de prioritate cu primul venit-primul servit în cazul în care prioritățile sunt egale. Totuși, mașina virtuală Java de pe Win32 folosește algoritmi de planificare specifici Win32.

Astfel, nu este posibil să prevedem ordinea în care firele de execuție vor obține un monitor în cazul în care mai multe fire de execuție sunt în așteptarea lui, și de aceea trebuie să evitați scrierea de cod care ar depinde de această ordine.

8.3.5 Sincronizarea metodelor statice

Am văzut deja că, pentru a putea fi executate, metodele care sunt declarate a fi sincronizate, sunt obligate să dețină monitorul obiectului pentru care au fost apelate. Ce putem însă spune atunci despre metodele statice, care, după cum știm, nu trebuie neapărat apelate în conjuncție cu un obiect (metodele statice sunt specifice claselor, și nu obiectelor)?

Soluția oferită de Java este simplă: în momentul în care se apelează o metodă statică sincronizată, aceasta va trebui să obțină un monitor special, care este asociat fiecărei clase în parte. Cu alte cuvinte, fiecare clasă Java are asociat un monitor care controlează accesul la metodele statice sincronizate ale clasei. Rezultă de aici că doar o singură metodă statică sincronizată se poate executa la un moment dat în cadrul unei clase.

Un fapt care merită remarcat este că, în implementarea actuală, mașina virtuală Java folosește pentru sincronizarea metodelor statice monitorul instanței

de tip `java.lang.Class` asociat clasei respective. Așa cum am văzut și în paragraful 5.8.2, pagina 165, instanțele clasei `java.lang.Class` descriu toate clasele și interfețele existente în cadrul unei aplicații Java aflată în execuție. Fiecare clasă sau interfață Java (chiar și tipurile primitive) are asociată o instanță de tip `java.lang.Class`. Obiectele de tip `Class` sunt construite automat de către mașina virtuală Java pe măsură ce clasele sunt încărcate în memorie. Prin intermediul unei instanțe `java.lang.Class` asociată unei clase se pot obține informații despre clasa respectivă, cum ar fi numărul de atribute și tipul lor, metodele clasei, pachetul din care clasa face parte etc. Instanța `java.lang.Class` asociată unei clase poate fi obținută folosind metoda `getClass()` definită chiar în clasa `Object`. Ajunși în acest punct, vă întrebați, probabil, la ce ne folosește să avem o clasă care să descrie conținutul fiecărei clase? Răspunsul este că această clasă este piatra de temelie a așa-numitului Reflection API (vezi capitolul 5.8.2), care este folosit în *debugger*-e⁵, serializare și deserializare de obiecte etc.

Pentru a ne convinge de faptul că metodele statice folosesc monitorul asociat instanței `java.lang.Class`, să studiem clasa `StaticMonitorTest` de mai jos:

```

1 public class StaticMonitorTest extends Thread
2 {
3     public void run()
4     {
5         synchronized ( getClass () )
6         {
7             System.out.println("Se executa run()");
8             try { sleep(5000); } catch (InterruptedException _) { }
9         }
10    }
11
12    public static synchronized void staticF()
13    {
14        System.out.println("Se executa staticF()");
15        try { sleep(5000); } catch (InterruptedException _) { }
16    }
17
18    public static void main(String[] args)
19    {
20        new StaticMonitorTest().start();
21        staticF();
22    }
23 }

```

La rularea acestei clase pe Linux sau Win32, se afișează mesajul "Se ex-

⁵aplicații folosite în depanarea programelor

`executa staticF()`”, după care urmează o pauză de 5 secunde, se afișează “Se executa `run()`” și programul se încheie după o pauză de încă 5 secunde. Rezultă clar de aici că monitorul folosit pentru metoda statică este același cu monitorul obiectului de tip `Class` asociat. Nu se știe dacă ne putem baza pe acest comportament și în versiunile viitoare de Java. Un lucru este totuși cert: două metode statice definite în cadrul aceleiași clase vor utiliza același monitor.

8.3.6 Monitoare și atribute publice

Am arătat deja că declararea de atribute publice nu este de dorit. Acest lucru devine și mai evident dacă privim problema din perspectiva accesului concurrent. Atributele publice ale unui obiect pot fi accesate de către un fir de execuție fără a beneficia de protecția oferită de metodele sincronizate. De fiecare dată când declarăm un atribut ca fiind public, lăsăm controlul asupra actualizării acelui atribut în seama oricărui programator care ne utilizează clasa, ceea ce poate conduce cu ușurință la inconsistență la concurență.

Observație: Programatorii Java obișnuiesc adeseori să declare constante simbolice sub forma unor atribute statice de tip `final public`. Atributele declarate în acest fel nu pun probleme la concurență, deoarece inconsistențele pot apărea doar în conjuncție cu obiecte a căror valoare nu este constantă.

8.3.7 Când NU trebuie să folosim sincronizarea

Ajunși în acest punct cu lectura acestui capitol ar trebui să fiți deja capabili să scrieți cod *thread-safe*⁶, folosind cuvântul cheie `synchronized`. Să vedem acum când este într-adevăr necesar să folosim metode sincronizate și care sunt dezavantajele folosirii unor astfel de metode.

Cel mai adesea, programatorii nu folosesc instrucțiunea `synchronized` deoarece scriu cod `single-threaded` (pe un singur fir de execuție). De exemplu, rutinele care solicită procesorul intensiv nu beneficiază prea mult de fire multiple de execuție. Un compilator nu are performanțe semnificativ mai bune dacă rulează pe mai multe fire de execuție. Ca dovadă, compilatorul Java de la Sun nu conține prea multe metode sincronizate, deoarece se consideră că rulează în special în propriul său fir de execuție, fără a fi necesar să partajeze resursele cu alte fire de execuție.

Un alt motiv comun pentru evitarea metodelor sincronizate este că ele nu sunt la fel de eficiente ca metodele nesincronizate. Anumite teste simple au pus

⁶cod care nu crează probleme prin execuția pe mai multe fire

în evidență faptul că metodele sincronizate sunt de aproximativ patru ori mai lente decât variantele lor nesincronizate. Aceasta nu implică neapărat faptul că întreaga aplicație va fi de patru ori mai lentă, dar este o problemă care trebuie avută în vedere. Există programe care necesită drămuirea fiecărui gram de utilizare din sistemul pe care rulează (de exemplu, programele Java care rulează pe PDA⁷-uri). În asemenea situații este adeseori adecvat să eliminăm surplusul de calcule generat de metodele sincronizate acolo unde este posibil.

Îmbunătățiri ale performanței pot să apară și dacă, în loc să declarăm o metodă întreagă ca fiind sincronizată, folosim un bloc `synchronized` doar pentru cele câteva instrucțiuni care trebuie să fie sincronizate. Aceasta se aplică mai ales atunci când avem metode lungi, care realizează multe calcule, și doar o mică porțiune din ele este susceptibilă de inconsistență la concurență.

8.3.8 Blocare circulară (deadlock)

Blocarea circulară, numită uneori și *îmbrățișare mortală* este unul dintre cele mai grave evenimente care pot să apară într-un mediu multithreaded. Programele Java nu sunt imune la blocări circulare (deadlocks), iar programatorii trebuie să fie atenți pentru a le evita.

O blocare circulară este o situație în care două sau mai multe fire de execuție sunt blocate în așteptarea unei resurse și nu pot continua execuția. În cazul cel mai simplu, avem două fire de execuție fiecare dintre ele așteptând eliberarea unui monitor deținut de celălalt fir. Fiecare fir este pus în așteptare, până când monitorul dorit devine disponibil. Totuși, acest lucru nu se va realiza niciodată. Primul fir așteaptă monitorul deținut de al doilea fir, iar al doilea așteaptă monitorul deținut de primul. Deoarece fiecare fir este în așteptare, nici unul nu va elibera monitorul necesar celuilalt fir.

Programul din **Listing 8.6** este un exemplu simplu menit să vă dea o idee despre cum poate să apară o blocare circulară.

Listing 8.6: Exemplu de blocare circulară

```
1 /* * Clasa care provoaca o blocare circulara prin
2 * pornirea a doua fire de executie care se asteapta reciproc. */
3 public class Deadlock implements Runnable
4 {
5     private Deadlock otherThread ;
```

⁷PDA = Personal Digital Assistant, cunoscute și sub numele de Handheld PC sau Palm, reprezintă unități mobile de dimensiune mică având capacitatea de a stoca și reține informație. Au fost folosite în special ca niște agende electronice, capabile să rețină adrese, întâlniri etc. Funcționalitatea lor s-a extins în prezent, fiind capabile de operații mai complexe, cum ar fi vizualizarea de documente, consultarea poștei electronice (email), calcul tabelar etc.

```

6  public static void main(String [] args)
7  {
8      Deadlock d1 = new Deadlock ();
9      Deadlock d2 = new Deadlock ();
10     Thread t1 = new Thread (d1);
11     Thread t2 = new Thread (d2);
12
13     d1.otherThread = d2;
14     d2.otherThread = d1;
15     t1.start ();
16     t2.start ();
17 }
18
19
20 public synchronized void run()
21 {
22     try { Thread.sleep (2000); } catch (InterruptedException e) { }
23     System.out.println ("Am intrat in metoda run");
24     otherThread.syncMethod ();
25 }
26
27 public synchronized void syncMethod ()
28 {
29     try { Thread.sleep (2000); } catch (InterruptedException e) { }
30     System.out.println ("Am intrat in metoda sincronizata");
31 }
32 }

```

În acest exemplu, metoda `main()` lansează în liniile 15-16 două fire de execuție, fiecare dintre ele apelând metoda sincronizată `run()` a clasei `Deadlock`. Când primul fir se trezește (după ce a dormit 2 secunde) la linia 23 afișează mesajul de intrare în metoda `run()`, după care încearcă să apeleze metoda `syncMethod()` a celuilalt fir de execuție la linia 24. Evident, monitorul celuilalt obiect este deținut de al doilea fir de execuție, așa că primul fir stă și așteaptă eliberarea monitorului. În momentul în care se trezește, al doilea fir încearcă să apeleze `syncMethod()` a primului fir. Cum monitorul primului fir este deja deținut de metoda `run()` a acestuia, și al doilea fir va intra în așteptare. Cele două fire de execuție vor aștepta unul după altul și nu își vor continua niciodată execuția. Acest fapt este demonstrat și de mesaje afișate la execuția programului:

```

Am intrat in metoda run
Am intrat in metoda run

```

după care programul se blochează fără să mai facă nimic, până când este întrerupt cu CTRL-BREAK.

Observație: Putem vedea exact ce se întâmplă în cadrul mașinii virtuale Java folosind următorul truc valabil pentru JDK pe Solaris și Linux (pe Windows nu funcționează): apăsați CTRL+\ în fereastra de terminal în care rulează aplicația Java. Aceasta are ca efect afișarea pe ecran a stării în care se află mașina virtuală. Iată o secvență din textul afișat de mașina virtuală după ce am apăsăm CTRL+\ la câteva secunde după lansarea aplicației Deadlock:

```
"Thread-1" prio=1 tid=0x80b2e00 nid=0x9b5 waiting
for monitor entry [0xbe7ff000, .0xbe7ff8b0]
at Deadlock.syncMethod (Deadlock.java:30)
at Deadlock.run (Deadlock.java:25)
at java.lang.Thread.run (Thread.java:484)
"Thread-0" prio=1 tid=0x80b2600 nid=0x9b4 waiting
for monitor entry [0xbe9ff000, .0xbe9ff8b0]
at Deadlock.syncMethod (Deadlock.java:30)
at Deadlock.run (Deadlock.java:25)
at java.lang.Thread.run (Thread.java:484)
```

Reiese de aici clar că ambele fire de execuție (Thread-1 și Thread-0) așteaptă eliberarea unui monitor în cadrul metodei run().

Există numeroși algoritmi pentru prevenirea și detectarea blocării circulare, dar prezentarea lor depășește cadrul acestui capitol (există multe manuale de baze de date și sisteme de operare care discută pe larg algoritmi de detectare a blocajelor circulare). Din păcate, mașina virtuală Java nu face nimic pentru a detecta blocajele circulare. Totuși specificațiile Java nu împiedică acest lucru, așa că nu este exclus ca această facilități să fie adăugată mașinii virtuale în versiunile viitoare.

8.4 Coordonarea firelor de execuție

În paragraful precedent am văzut modul în care monitorul unui obiect poate fi utilizat pentru a serializa accesul la o anumită secvență de cod. Totuși, monitoarele sunt mai mult decât niște obiecte de blocare (lock), deoarece ele pot fi utilizate și pentru a coordona execuția mai multor fire de execuție folosind metodele wait() și notify().

8.4.1 De ce este necesar să coordonăm firele de execuție?

Firele de execuție din cadrul unui program Java sunt adeseori interdependente; un fir de execuție poate să depindă de un alt fir de execuție care trebuie să încheie o anumită operație sau să satisfacă o cerere. De exemplu, un program

de calcul tabelar poate să realizeze un calcul care solicită intensiv procesorul pe un fir de execuție separat. Dacă un fir de execuție atașat interfeței utilizator încearcă să actualizeze imaginea afișată de program, acesta va trebui să se coordoneze cu firul care realizează calculul, începând actualizarea imaginii doar în momentul în care firul de calcul s-a încheiat.

Există foarte multe situații în care este util să coordonăm două sau mai multe fire de execuție. Lista care urmează identifică unele dintre situațiile mai uzuale:

- Zonele tampon (buffere) partajate sunt adeseori utilizate pentru a transmite date între firele de execuție. În această situație, există de obicei un fir de execuție care scrie în buffer (numit *writer* sau *producător*) și un fir care citește din buffer (numit *reader* sau *consumator*). Când firul consumator citește din buffer, trebuie să se coordoneze cu firul producător, citind date din buffer doar *după* ce acestea au fost plasate acolo de producător. Dacă bufferul este gol, firul consumator trebuie să aștepte noi date (fără să verifice încontinuu dacă au venit date!). Firul producător trebuie să-l notifice pe consumator când a scris ceva în buffer, pentru ca acesta să poată să continue citirea;
- Dacă o aplicație trebuie să răspundă cu promptitudine la acțiunile utilizatorului, dar ocazional trebuie să realizeze anumite calcule numerice intensive, putem rula firul de calcul numeric cu o prioritate mică (folosind metoda `setPriority()` din clasa `Thread`). Toate firele de execuție de prioritate mai mare care au nevoie de rezultatul calculului vor aștepta ca firul cu prioritate mai mică să se încheie, moment în care acesta va notifica toate firele de execuție interesate că și-a încheiat calculul;
- Un fir de execuție poate fi construit în așa manieră încât execută anumite operații ca răspuns la anumite evenimente generate de alte fire de execuție. Dacă nu există nici un eveniment netratat, firul este suspendat (un fir de execuție care nu face nimic nu ar trebui să consume procesor). Firele de execuție care generează evenimente trebuie să aibă la dispoziție un mecanism prin care să notifice firul care așteaptă de faptul că s-a produs un eveniment.

Nu este întâmplător faptul că exemplele de mai sus au folosit în mod repetat cuvintele “așteaptă” și “notifică”. Aceste două cuvinte exprimă conceptele care stau la baza coordonării firelor de execuție: un fir așteaptă producerea unui eveniment, iar un alt fir notifică producerea acelui eveniment. Aceste cuvinte, `wait` (așteaptă) și `notify` (notifică) sunt folosite și în Java, ca numele metodelor care se apelează pentru a coordona firele de execuție (`wait()` și `notify()` din clasa `Object`).

Așa cum am arătat în paragraful 8.3.1 din cadrul acestui capitol, fiecare obiect Java are asociat un monitor. Acest fapt devine foarte util în acest moment, deoarece vom vedea că monitoarele sunt folosite și pentru a implementa primitivele de coordonare a firelor de execuție. Deși monitoarele nu sunt accesibile în mod direct programatorului, clasa `Object` furnizează două metode prin intermediul cărora putem interacționa cu monitorul unui obiect. Aceste două metode sunt `wait()` și `notify()`.

8.4.2 `wait()` și `notify()`

Firele de execuție sunt de obicei coordonate folosind conceptul de *condiție* sau *variabilă condițională*. O condiție reprezintă o stare sau un eveniment fără de care firul de execuție nu poate continua execuția. În Java, acest model este exprimat în general sub forma

```
1 while ( ! conditia_dupa_care_astept )
2 {
3     wait () ;
4 }
```

La început, se verifică dacă variabila condițională este adevărată. Dacă da, nu este necesar să se aștepte. În cazul în care condiția nu este încă adevărată, se apelează metoda `wait()`. Când așteptarea (`wait()`) se încheie (fie pentru că firul a fost notificat, fie pentru că a expirat timpul de așteptare) se verifică din nou condiția pentru a ne asigura că a devenit adevărată (dacă suntem siguri că a devenit adevărată, putem înlocui ciclul `while` cu un simplu `if`).

Apelul metodei `wait()` pe un obiect oprește temporar firul de execuție curent până când un alt fir apelează `notify()`, pentru a informa firul care așteaptă că s-a produs un eveniment care ar putea schimba condiția de așteptare. Cât timp un fir de execuție așteaptă în cadrul unei instrucțiuni `wait()`, acesta este considerat de către mașina virtuală ca fiind suspendat și nu i se alocă timp de execuție pe procesor până în momentul în care este trezit printr-un apel al lui `notify()` dintr-un alt fir de execuție. Apelul lui `notify()` trebuie făcut dintr-un alt fir, deoarece firul curent este suspendat, deci nu are cum să apeleze `notify()`. Apelul lui `notify()` va informa un singur fir de execuție aflat în așteptare de faptul că starea obiectului s-a modificat, încheind astfel așteptarea din cadrul lui `wait()`.

Metoda `wait()` are și două variațiuni ușor diferite. Prima versiune acceptă un parametru de tip `long`, interpretat ca perioadă de expirare a așteptării (timeout) măsurată în milisecunde. A doua variantă acceptă doi parametri, interpretați tot ca o perioadă de expirare (în milisecunde și nanosecunde) a așteptării. Aceste variante sunt folosite atunci când nu dorim să așteptăm la infinit

producerea unui anumit eveniment. Dacă dorim să renunțăm la așteptare după o anumită perioadă, va trebui să folosim una dintre metodele:

```
wait( long milliseconds )
wait( long milliseconds , int nanoseconds )
```

Un mic inconvenient este dat de faptul că aceste metode nu oferă o modalitate prin care să determinăm dacă apelul lui `wait()` s-a încheiat printr-un apel de `notify()` sau prin expirare (timeout). Totuși această problemă poate fi ușor rezolvată, deoarece putem verifica din nou condiția de așteptare și timpul curent pentru a determina care dintre cele două evenimente s-a produs.

Metodele `wait()` și `notify()` pot fi apelate doar dacă firul de execuție curent deține monitorul obiectului pe care sunt apelate. Vom detalia acest aspect în paragrafele următoare.

8.4.3 Exemplu de coordonare a firelor de execuție: problema consumator-producător

Exemplul clasic de coordonare a firelor de execuție, folosit în majoritatea lucrărilor de programare este problema bufferului de capacitate limitată. În varianta sa cea mai simplă, această problemă presupune că se utilizează un buffer (zonă tampon de memorie) pentru a comunica între două fire de execuție sau procese (în multe sisteme de operare, bufferele de comunicare între procese au dimensiune fixă). Există un fir de execuție care scrie date în buffer și un altul care preia datele din buffer spre a fi prelucrate mai departe. Pentru ca bufferul să funcționeze corect, trebuie ca firul care scrie și cel care citește să fie coordonate, astfel încât:

- firul care scrie poate să adauge date în buffer până când acesta se umple, moment în care firul este suspendat;
- după ce firul care citește extrage date din bufferul plin, notifică firul care scrie asupra stării modificate a bufferului, care este activat și i se permite să reia scrierea;
- firul care citește poate să extragă date din buffer până când acesta se golește, moment în care firul este suspendat;
- când firul care scrie adaugă date la bufferul gol, firul care citește este notificat asupra stării modificate a bufferului, este activat și i se permite să reia citirea.

Vom prezenta în continuare trei clase Java care rezolvă problema bufferului de capacitate limitată. Acestea sunt denumite în mod clasic *Producer* (clasa care implementează firul de execuție care scrie în buffer), *Consumer* (clasa care implementează firul de execuție care citește din buffer) și *Buffer* (clasa centrală, care oferă metode pentru a adăuga și citi date în și din buffer).

Să începem cu clasa *Producer*, care este extrem de simplă:

```

1 public class Producer implements Runnable
2 {
3     private Buffer buffer;
4
5     public Producer(Buffer buffer)
6     {
7         this.buffer = buffer ;
8     }
9
10    public void run()
11    {
12        for (int i=0; i<250; i++)
13        {
14            buffer.put((char)('A' + (i%26)));
15        }
16    }
17 }

```

Clasa *Producer* implementează interfața *Runnable*, ceea ce reprezintă un indiciu al faptului că va rula pe un fir de execuție separat. În momentul în care se apelează metoda *run()* a clasei *Producer*, se va scrie rapid în *Buffer* o succesiune de 250 de caractere. Dacă clasa *Buffer* nu este capabilă să rețină toate cele 250 de caractere, rămâne în responsabilitatea metodei *put()* a bufferului să realizeze coordonarea adecvată a firelor de execuție (vom vedea imediat cum).

Clasa *Consumer* este la fel de simplă ca și clasa *Producer*:

```

1 public class Consumer implements Runnable
2 {
3     private Buffer buffer;
4
5     public Consumer(Buffer b)
6     {
7         buffer = b;
8     }
9
10    public void run()
11    {
12        for (int i=0; i<250; i++)
13        {
14            System.out.println(buffer.get());

```

246

```

15     }
16 }
17 }

```

Ca și în cazul clasei `Producer`, clasa `Consumer` implementează de asemenea interfața `Runnable`. Metoda ei `run()` citește cu lăcomie 250 de caractere din `Buffer`. În situația în care `Consumer` încearcă să preia caractere dintr-un `Buffer` gol, metoda `get()` a clasei `Buffer` este responsabilă de a coordona firul de execuție al consumatorului.

Iată acum care este codul pentru clasa `Buffer`. Două dintre metodele ei, `get()` și `put()`, au fost deja amintite.

Listing 8.7: Clasa `Buffer`

```

1 public class Buffer
2 {
3     private char[] buf ;//retine elementele din buffer
4     private int last ; //ultima pozitie ocupata
5
6     /** Construiește un buffer cu size elemente */
7     public Buffer(int size)
8     {
9         buf = new char[size];
10        last = 0;
11    }
12
13    /** Intoarce true daca bufferul e plin.*/
14    public boolean isFull ()
15    {
16        return (last == buf.length);
17    }
18
19    /** Intoarce true daca bufferul e gol. */
20    public boolean isEmpty ()
21    {
22        return (last == 0);
23    }
24
25    /** Aadauga caracterul c la buffer. Daca bufferul
26     * este plin, atunci se asteapta pana cand se obtine
27     * o notificare de la metoda get(). */
28    public synchronized void put(char c)
29    {
30        while(isFull())
31        {
32            try
33            {
34                wait();
35            }

```

```

36     catch (InterruptedException e) { }
37 }
38 buf[ last++ ] = c;
39 // anunta firul consumator ca bufferul nu mai e gol
40 notify ();
41 }
42
43 /** Extrage primul caracter din Buffer. Daca bufferul
44  * este gol, se asteapta pana cand se obtine o notificare
45  * de la metoda get(). */
46 public synchronized char get()
47 {
48     while (isEmpty ())
49     {
50         try
51         {
52             wait ();
53         }
54         catch (InterruptedException e) { }
55     }
56     char c = buf[0];
57     //deplaseaza elementele din buffer la stanga
58     System.arraycopy (buf, 1, buf, 0, --last);
59     //anunta firul producator ca bufferul nu mai e plin
60     notify ();
61     return c;
62 }
63 }

```

Observație: Cititorii atenți, aflați la prima confruntare cu metodele `wait()` și `notify()` ar putea observa o contradicție. S-a menționat deja faptul că pentru a putea apela metodele `wait()` și `notify()`, firul de execuție trebuie să dețină monitorul obiectului. Astfel, dacă un fir de execuție obține monitorul unui obiect și intră apoi în așteptare printr-un apel al metodei `wait()`, cum va putea alt fir de execuție să obțină monitorul obiectului pentru a putea notifica primul fir? Oare monitorul nu este încă în posesia firului care așteaptă, împiedicând astfel al doilea fir în a-l obține? Răspunsul la acest aparent paradox este dat de modul în care este implementată în Java metoda `wait()`: aceasta eliberează temporar monitorul după ce a fost apelată, și obține din nou monitorul înainte de a se încheia. Astfel, metoda `wait()` permite și altor fire de execuție să obțină monitorul obiectului și (eventual) să apeleze metoda `notify()`.

Clasa `Buffer` este doar o simplă zonă tampon de memorie și nimic mai mult. Se pot adăuga elemente în buffer, folosind metoda `put()` și se pot extrage elemente din buffer folosind metoda `get()`.

Observați modul în care s-au utilizat `wait()` și `notify()` în aceste metode. În cadrul metodei `put()`, atâta timp cât bufferul este plin, se intră în așteptare. Astfel, nu se vor mai putea adăuga noi elemente în buffer până în momentul în care firul producător nu este notificat (la linia 60) de către consumator că au fost extrase elemente din buffer. Apelul metodei `notify()` din finalul metodei `get()` are rolul de a activa firul de execuție care așteaptă în cadrul metodei `put()` la linia 34 (dacă acesta există), permițându-i astfel să adauge un nou caracter în `Buffer`. Un raționament absolut similar este valabil și pentru metoda `get()`, în care se așteaptă cât timp bufferul este gol, după care se extrage un caracter și se notifică noua stare a bufferului.

Clasa `BufferTest` de mai jos, construiește un `Buffer`, după care pornește pe fire de execuție separate producătorul și consumatorul:

```
1 public class TestBuffer
2 {
3     public static void main(String args[])
4     {
5         Buffer b = new Buffer( 50 );
6         new Thread( new Producer( b ) ).start();
7         new Thread( new Consumer( b ) ).start();
8     }
9 }
```

Rezultatul afișat la consolă (în cadrul clasei `Consumer`) confirmă faptul că firele de execuție au fost corect coordonate:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z A B C D E F
```

Implementarea metodei `get()` din clasa `Buffer` nu este foarte eficientă. La fiecare apel al acestei metode, toate celelalte caractere din buffer trebuie deplasate către stânga. O problemă de la finalul acestui capitol propune o implementare mai eficientă a clasei `Buffer` folosind o coadă alocată static (detalii despre cozi în cel de-al doilea volum, dedicat structurilor de date și algoritmilor).

8.4.4 Deținerea monitoarelor

Există o restricție importantă în folosirea metodelor `wait()` și `notify()`: aceste metode pot fi apelate doar dacă firul de execuție curent deține monitorul obiectului. În multe situații, metodele `wait()` și `notify()` sunt apelate în cadrul unor metode sincronizate (deținerea monitorului fiind astfel asigurată), ca în exemplul de mai jos:

```
1 public synchronized void method()
2 {
3     ...
4     while (!condition)
5     {
6         wait ();
7     }
8     ...
9 }
```

În situația de mai sus, modificatorul `synchronized` ne asigură de faptul că firul de execuție care apelează `wait()` deține deja monitorul în momentul apelului.

Ce se întâmplă însă dacă se încearcă apelul `wait()` sau `notify()` fără a deține monitorul obiectului (de exemplu, în cadrul unei metode nesincronizate)? Compilatorul nu poate să își dea seama dacă există un apel ilegal (fără a deține monitorul) al acestor metode (de exemplu se poate ca o metodă nesincronizată să apeleze `wait()`, dar această metodă să fie apelată de o metodă sincronizată). Așadar, singura posibilitate este ca, în timpul execuției, dacă se întâlnește un apel al metodelor `wait()` sau `notify()` fără a deține monitorul obiectului să se genereze o excepție. Această excepție se numește `IllegalMonitorStateException` și este aruncată de mașina virtuală Java ori de câte ori întâlnește un apel ilegal al metodelor `wait()` sau `notify()`. Clasa `MonitorTest` de mai jos evidențiază ce se petrece în momentul în care se apelează metoda `wait()` fără a fi deținut în prealabil monitorul clasei:

```
1 public class MonitorTest
2 {
3     public static void main(String [] args)
4     {
5         MonitorTest mt = new MonitorTest();
6         mt.method();
7     }
8
9     public void method()
10    {
11        try
12        {
13            wait();
```

250

```

14     }
15     catch (InterruptedException e)
16     {
17     }
18 }
19 }

```

Dacă veți încerca să executați această clasă, veți primi următorul mesaj:

```

Exception in thread "main"
    java.lang.IllegalMonitorStateException
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:420)
    at MonitorTest.method(MonitorTest.java:13)
    at MonitorTest.main(MonitorTest.java:6)

```

Pentru a evita o astfel de excepție, este necesar ca de fiecare dată când apelezi metoda `wait()` pe un obiect să vă asigurați că firul de execuție deține monitorul acelui obiect.

8.4.5 Utilizarea instrucțiunii `synchronized`

Instrucțiunea `synchronized` poate fi și ea folosită pentru a realiza coordonarea firelor de execuție. În cele mai multe cazuri utilizarea metodelor sincronizate este suficientă, și datorită faptului că sintaxa lor este mai simplă ele sunt de preferat instrucțiunii `synchronized`. Există totuși două situații în care utilizarea metodelor sincronizate nu este de dorit, sau nu poate rezolva problema.

Dacă metoda care conține un `wait()` sau `notify()` este foarte lungă (și necesită un timp de execuție semnificativ) este preferabil să se sincronizeze doar secvența de cod care este strict necesară pentru a menține consistența la concurență.

Adeseori este necesar să coordonăm metode din obiecte diferite. În această situație, utilizarea de metode sincronizate nu are nici un sens, deoarece apelul lui `notify()` din cadrul unei metode sincronizate a unui obiect nu are nici un efect asupra firului de execuție aflat în așteptare în cadrul unui `wait()` pe un alt obiect (metodele obțin monitoare distincte)!

Să presupunem de exemplu că avem la dispoziție o clasă simplă `Buffer1` identică din punct de vedere al interfeței cu clasa `Buffer` din paragraful 8.4.3, dar care a fost concepută și scrisă pentru a funcționa corect pe un singur fir de execuție:

```

1 public class Buffer1
2 {
3     private char[] buf; //retine elementele din buffer
4     private int last; //ultima pozitie ocupata
5
6     /** Construiește un buffer cu size elemente */
7     public Buffer1(int size)
8     {
9         buf = new char[size];
10        last = 0;
11    }
12
13    /** Intoarce true daca bufferul e plin.*/
14    public boolean isFull()
15    {
16        return (last == buf.length);
17    }
18
19    /** Intoarce true daca bufferul e gol. */
20    public boolean isEmpty()
21    {
22        return (last == 0);
23    }
24
25    /** Adauga caracterul c la buffer. */
26    public synchronized void put(char c)
27    {
28        buf[last++] = c;
29    }
30
31    /** Extrage primul caracter din Buffer. */
32    public synchronized char get()
33    {
34        char c = buf[0];
35        //deplaseaza elementele din buffer la stanga
36        System.arraycopy(buf, 1, buf, 0, --last);
37        return c;
38    }
39 }

```

Observați că, spre deosebire de clasa `Buffer` din paragraful 8.4.3, această clasă nu conține nici o linie de cod pentru a coordona în vreun fel producătorul și consumatorul. Rămâne astfel în sarcina producătorului și a consumatorului să își coordoneze accesul la buffer. Având în vedere că producătorul și consumatorul sunt implementate de clase diferite, utilizarea metodelor sincronizate în vederea coordonării nu este posibilă. Coordonarea ar trebui să se facă folosind `wait()` și `notify()` pe același obiect. Care obiect? Chiar instanța clasei `Buffer1`. Codul de mai jos descrie clasa `Producer1`, care este o variantă

modificată a clasei `Producer`:

```

1 public class Producer1 implements Runnable
2 {
3     private Buffer1 buffer;
4
5     public Producer1(Buffer1 buffer)
6     {
7         this.buffer = buffer ;
8     }
9
10    public void run()
11    {
12        for (int i=0; i<250; i++)
13        {
14            synchronized ( buffer )
15            {
16                while ( buffer.isFull() )
17                {
18                    try
19                    {
20                        buffer.wait() ;
21                    }
22                    catch( InterruptedException e )
23                    {}
24                }
25                buffer.put((char)( 'A' + (i%26)));
26                buffer.notify() ;
27            }
28        }
29    }
30 }

```

Diferența față de clasa `Producer` apare la metoda `run()`, care conține cod suplimentar pentru a realiza sincronizarea cu consumatorul. În primul rând, la linia 14 se solicită monitorul bufferului, folosind instrucțiunea

```

synchronized (buffer)
{
    ...
}

```

Instrucțiunea `synchronized` este utilizată în acest caz din două motive. În primul rând, trebuie asigurată consistența la concurență, prin evitarea situației de a se scrie și citi în buffer în același timp. În al doilea rând, instrucțiunea `synchronized` garantează obținerea monitorului obiectului `buffer`, pe care se va putea apoi apela `wait()` în cazul în care bufferul este plin (linia 20), respectiv `notify()` după ce s-a citit din buffer (linia 26). Remarcați faptul că `wait()` și `notify()` au fost apelate în conjuncție cu obiectul `buffer`:

```

buffer.wait()
buffer.notify()

```

deoarece coordonarea se face utilizând monitorul bufferului.

Implementarea consumatorului este similară cu cea a producătorului și este realizată de clasa `Consumer1` de mai jos:

```

1 public class Consumer1 implements Runnable
2 {
3     private Buffer1 buffer;
4
5     public Consumer1(Buffer1 buffer)
6     {
7         this.buffer = buffer ;
8     }
9
10    public void run()
11    {
12        for (int i=0; i<250; i++)
13        {
14            synchronized ( buffer )
15            {
16                while ( buffer.isEmpty() )
17                {
18                    try
19                    {
20                        buffer.wait() ;
21                    }
22                    catch( InterruptedException e )
23                    {}
24                }
25                System.out.print( buffer.get() + "-" );
26                buffer.notify() ;
27            }
28        }
29    }
30 }

```

Pentru a testa noua rezolvare a problemei bufferului, am creat clasa `TestBuffer1`, absolut similară clasei `TestBuffer`:

```

1 public class TestBuffer1
2 {
3     public static void main(String args[])
4     {
5         Buffer1 b = new Buffer1 ( 2 ) ;
6         new Thread( new Producer1( b ) ).start() ;
7         new Thread( new Consumer1( b ) ).start() ;
8     }
9 }

```

La execuția acestei clase, se va afișa aceeași secvență de caractere ca și la rezolvarea precedentă, ceea ce dovedește că și această abordare este viabilă.

8.4.6 Un alt exemplu: consumatori și producători multipli

Este posibil ca mai multe fire de execuție să fie în așteptare (`wait()`) pe același obiect. Această situație poate să apară dacă mai multe fire de execuție așteaptă apariția aceluiași eveniment, sau dacă există mai multe fire de execuție care partajează o resursă unică. Să luăm din nou exemplul clasei `Buffer` prezentată în secțiunea 8.4.3. `Buffer`ul era utilizat de către un singur producător și un singur consumator. Ce s-ar petrece însă, dacă ar exista mai mulți producători? Dacă `buffer`ul se umple, ar putea exista mai mulți producători care să încerce să adauge date în `Buffer`. Toți aceștia ar sta blocați în cadrul metodei `put()`, așteptând ca un consumator să elibereze ceva spațiu în `Buffer`.

În momentul în care se apelează `notify()` pe un obiect, pot exista zero, unul sau mai multe fire de execuție blocate în cadrul unui `wait()` pe acel obiect. Dacă nu există nici un fir de execuție în așteptare, apelul metodei `notify()` nu are nici un efect. Dacă un singur fir de execuție este în așteptare, acesta va fi notificat și va începe să aștepte eliberarea monitorului de către firul de execuție care a apelat `notify()`. Dacă există două sau mai multe fire de execuție în așteptare, mașina virtuală Java va alege un singur fir de execuție, pe care îl va notifica.

Cum se face alegerea firului de execuție care va fi notificat? Situația este aceeași cu cea în care mai multe fire de execuție așteaptă eliberarea unui monitor: comportamentul nu este specificat. Totuși, implementările curente ale mașinii virtuale Java, folosesc un algoritm bine definit. Astfel, mașina virtuală de pe Solaris va alege firul de execuție cu prioritatea cea mai mare și îl va notifica. Dacă există mai mult de un fir de execuție cu aceeași prioritate, va fi ales firul care a intrat primul în `wait()`. Situația în cazul mașinii virtuale de pe Windows este ceva mai complicată, deoarece în acest caz se folosesc algoritmi de planificare specifici sistemului de operare.

Astfel, deși se poate prezice care dintre fire de execuție va fi notificat, nu trebuie în nici un caz să scriem cod care să se bazeze pe astfel de deducții. Singurul fapt pe care ne putem baza este că doar un singur fir de execuție va fi notificat la apelul lui `notify()` (dacă există cel puțin un fir de execuție în așteptare).

Utilizarea metodei `notifyAll()`

Există situații în care vom dori notificarea fiecărui fir de execuție care așteaptă la un moment dat pe un obiect. Interfața clasei `Object` oferă o metodă care face exact acest lucru: `notifyAll()`. În timp ce metoda `notify()` trezește un singur fir de execuție, metoda `notifyAll()` va trezi toate firele de execuție aflate în acel moment în așteptare pe monitorul obiectului.

Un exemplu concret în care este necesară utilizarea lui `notifyAll()` îl constituie problema bufferului cu producători și consumatori multipli. În versiunea prezentată în cadrul paragrafului 8.4.3, clasa `Buffer` utilizează metoda `notify()` pentru a trimite o notificare unui singur fir de execuție aflat în așteptare pe un buffer plin sau gol. Totuși, în practică, nu există nici o garanție că doar un singur fir de execuție se află în așteptare, deoarece pot exista consumatori și producători multipli. Iată o altă variantă a clasei `Buffer` denumită (din lipsă de imaginație) `Buffer2`, care folosește `notifyAll()` și este capabilă să coordoneze consumatori și producători multipli.

Listing 8.8: Clasa `Buffer2`

```

1 public class Buffer2
2 {
3     private char[] buf; //contine elementele din buffer
4     private int last = 0; // ultima pozitie ocupata
5     private int writersWaiting = 0; //nr. fire ce asteapta in put()
6     private int readersWaiting = 0; //nr. fire ce asteapta in get()
7
8     /** Construiește un buffer cu size elemente */
9     public Buffer2(int size)
10    {
11        buf = new char[size];
12    }
13
14    /** Intoarce true daca bufferul e plin.*/
15    public boolean isFull()
16    {
17        return (last == buf.length);
18    }
19
20    /** Intoarce true daca bufferul e gol. */
21    public boolean isEmpty()
22    {
23        return (last == 0);
24    }
25
26    /** Aadauga caracterul c la buffer. Daca bufferul este plin,
27     * atunci se incrementeaza <code>writersWaiting</code>
28     * si se asteapta pana cand se obtine

```

```

29  * o notificare de la metoda get(). */
30  public synchronized void put(char c)
31  {
32      while( isFull() )
33      {
34          try
35          {
36              writersWaiting++;
37              wait();
38          }
39          catch( InterruptedException e )
40          { }
41          writersWaiting--;
42      }
43      buf[ last++ ] = c;
44      if ( readersWaiting > 0 )
45      {
46          notifyAll();
47      }
48  }
49
50  /** Extrage primul caracter din Buffer. Daca bufferul
51  * este gol, se incrementeaza readersWaiting si se
52  * asteapta pana cand se obtine o notificare
53  * de la metoda get(). */
54  public synchronized char get()
55  {
56      while( isEmpty() )
57      {
58          try
59          {
60              readersWaiting ++ ;
61              wait();
62          }
63          catch( InterruptedException e )
64          { }
65          readersWaiting--;
66      }
67      char c = buf[0];
68      System.arraycopy( buf, 1, buf, 0, --last );
69      if ( writersWaiting > 0 )
70      {
71          notifyAll();
72      }
73      return c;
74  }
75 }

```

După cum ați observat, metodele `get()` și `put()` au fost făcute mai inteligente. Ele verifică mai întâi dacă este necesară vreo notificare (liniile 44 și

69), după care utilizează `notifyAll()` pentru a difuza notificarea la toate firele de execuție aflate în așteptare.

8.4.7 Utilizarea lui `InterruptedException`

Ați observat cu siguranță utilizarea consecventă de-a lungul acestui capitol a clasei `InterruptedException`. Dacă veți examina modul în care este declarată metoda `wait()` în clasa `Object`, veți înțelege de ce:

```
public final void wait() throws InterruptedException
```

`wait()` declară că ar putea să arunce o `InterruptedException`. Această excepție se generează atunci când se apelează metoda `interrupt()` a clasei `Thread`. Limbajul Java oferă posibilitatea de a întrerupe așteptarea de orice natură (*wait*, *sleep*) a unui fir de execuție prin generarea unei excepții de tipul `InterruptedException` într-un alt fir de execuție, folosind metoda `interrupt()`. Această metodă este utilizată pentru a trezi firele de execuție aflate în așteptare în cadrul unui `wait()`, `sleep()` sau a altor operații (de exemplu citire de la un socket).

Rezumat

Este deosebit de important să distingem situațiile în care este cazul să se utilizeze mai multe fire de execuție de situațiile în care trebuie să le evităm. Cel mai important motiv pentru utilizarea firelor de execuție este necesitatea de a realiza mai multe operații, a căror rulare amalgamată va produce o utilizare mai eficientă a mașinii de calcul (incluzând aici și posibilitatea de a distribui operațiile pe mai multe procesoare) sau va îmbunătăți interacțiunea cu utilizatorul.

Totuși, utilizarea firelor de execuție are și dezavantaje, cum ar fi scăderea vitezei din cauza așteptării după resurse partajate și creșterea utilizării procesorului pentru a le gestiona.

Un alt avantaj important al firelor de execuție este că, așa cum am menționat la începutul capitolului, ele înlocuiesc schimbările de context mari consumatoare de resurse generate de procese (de ordinul miilor de instrucțiuni) cu schimbări de context rapide (de ordinul sutelor de instrucțiuni). Având în vedere faptul că toate firele de execuție din cadrul unui proces partajează aceeași zonă de memorie, o schimbare de context nu implică în acest caz decât modificarea punctului de execuție și a variabilelor locale. Pe de altă parte, o schimbare de context în cazul proceselor implică interschimbarea întregului spațiu de memorie.

Firele de execuție reprezintă o lume nouă, iar a învăța să le utilizezi este echivalent cu asimilarea unui nou limbaj de programare. Una dintre principalele dificultăți legată de firele de execuție este dată de faptul că mai multe fire pot partaja aceleași resurse, cum ar fi atributele unui obiect, iar în acest caz trebuie să vă asigurați că nu se poate întâmpla ca mai multe fire să încerce să acceseze resursa în același timp. Aceasta presupune utilizarea atentă a cuvântului cheie *synchronized*, care este o unealtă extrem de utilă, dar care trebuie înțeleasă în profunzime, deoarece poate să provoace situații subtile de blocaj circular.

Capitolul de față este departe de a epuiza toate subiectele legate de firele de execuție. Nu am vorbit aici despre prioritatea firelor de execuție, despre grupuri de fire de execuție și clasa *ThreadGroup* sau despre fire de execuție *daemon*. Veți găsi informații despre toate aceste subiecte în excelenta carte a lui Bruce Eckel, *Thinking in Java* (vezi [Eckel]). Pentru o prezentare mai avansată a firelor de execuție, consultați *Concurrent Programming in Java*, de Doug Lea, Addison-Wesley, 1997.

Noțiuni fundamentale

blocaj circular (deadlock): situație în care două sau mai multe fire de execuție așteaptă eliberarea a două sau mai multe resurse pe care le dețin reciproc (deci nu vor fi niciodată eliberate).

coordonare: termen utilizat pentru a desemna dirijarea execuției a două sau mai multe fire de execuție folosind metodele `wait()` și `notify()`.

fir de execuție (thread): un flux de execuție în cadrul unui proces.

inconsistență la concurență (race condition): eroare de programare care cauzează comportamentul eronat sau imprevizibil al unui program. Mai exact, această eroare apare în cazul în care rezultatul unei operații depinde de factori temporali imprevizibili, cum ar fi ordinea în care s-a alocat timp pentru execuția firelor de execuție.

monitor: obiect special utilizat pentru a realiza excluderea reciprocă a unui grup de metode.

partajare de resurse: două sau mai multe fire de execuție au acces la aceleași resurse (de exemplu, la atributele unui obiect).

proces: program de sine stătător, care dispune de propriul lui spațiu de adrese.

serializare: constă în a impune accesul unui singur fir de execuție la un moment dat la o resursă în același timp.

sincronizare: reprezintă coordonarea unor evenimente astfel încât doar un anumit eveniment se produce la un moment dat.

Erori frecvente

1. Cea mai frecventă eroare este că programatorul nu este conștient că rulează într-un mediu multithreaded (de exemplu în cazul unui servlet Java) și scrie cod ignorând complet inconsistențele la concurență.
2. Mulți începători definesc firele de execuție corect, dar uită să le pornească.
3. Din cauza grabei sau a lipsei de experiență, adeseori nu sunt detectate secvențele de cod care pot conduce la inconsistență la concurență.
4. Nu este suficient să sincronizăm modificatorii unui atribut pentru a obține consistența la concurență. Trebuie sincronizate și secvențele de cod care accesează acel atribut.
5. Apelarea metodei `wait()` pe alt obiect decât cel pe care s-a făcut sincronizarea va genera o `IllegalMonitorStateException`.
6. Apelul lui `notify()` pe un obiect pentru a notifica un fir de execuție care se află în așteptare pe un alt obiect, nu va avea nici un efect.

Exerciții

Pe scurt

1. Care este diferența între fire de execuție și procese?
2. Ce interfață Java trebuie implementată de către firele de execuție?
3. Ce metodă este apelată pentru a porni un fir de execuție?
4. Ce metodă este apelată de către mașina virtuală la pornirea unui fir de execuție?
5. Când trebuie utilizat cuvântul cheie `synchronized`?
6. Când este de preferat instrucțiunea `synchronized()` metodelor sincronizate?
7. Ce este un monitor?
8. Care este diferența între coordonarea și sincronizarea firelor de execuție?

În practică

1. Creați două clase derivate din `Thread`. Metoda `run()` a primei clase obține o referință către cea de-a doua clasă, după care apelează `wait()` pe acest obiect. Metoda `run()` a celei de-a doua clase va apela metoda `notifyAll()` pentru primul fir după câteva secunde, astfel încât acesta să poată afișa un mesaj.
2. Construiți o clasă `AccountTest` care să pună în evidență problemele de inconsistență la concurență pe care le are clasa `Account` din paragraful 8.3.2.
3. Modul în care este implementată metoda `get()` în clasa `Buffer` din paragraful 8.4.3 este inefficient, deoarece necesită deplasarea tuturor elementelor din buffer către stânga (având astfel complexitatea $O(n)$). Modificați clasa `Buffer` pentru a reține elementele sub forma unei cozi (detalii despre cozi în volumul al doilea, dedicat structurilor de date și algoritmilor) astfel încât metodele `put()` și `get()` să aibă complexitatea $O(1)$.
4. Clasele `Buffer1`, `Consumer1` și `Producer1` din paragraful 8.4.5 constituie o implementare a problemei bufferului cu un singur producător și un singur consumator în care sincronizarea și coordonarea firelor de execuție este făcută în cadrul producătorului și consumatorului. Adaptați această soluție pentru a funcționa și în cazul problemei bufferului cu producători și consumatori multipli fără a modifica clasa `Buffer`!

Indicație

Se vor adăuga două atribute statice în cadrul claselor `Producer` și `Consumer`, `writersWaiting` respectiv `readersWaiting`, care vor fi incrementate și decrementate de câte ori un fir intră în așteptare sau este trezit. Pentru a asigura consistența la concurență, trebuie să se definească metode statice sincronizate care să incrementeze respectiv să decrementeze aceste variabile. Coordonarea consumatorilor și producătorilor se va face utilizând `wait()` și `notifyAll()`, care vor fi apelate pe monitorul clasei `Buffer` (ca în paragraful 8.4.5).

5. Problema damelor constă în a așeza n dame pe o tablă de șah de dimensiune $n \times n$, astfel încât acestea să nu se atace între ele. Rezolvați această problemă astfel încât pentru fiecare posibilitate de a așeza o damă pe o coloană în cadrul unei linii să se pornească un fir de execuție separat

(practic, se vor încerca în paralel toate variantele de a așeza dama pe o anumită line)!

Proiecte de programare

1. Scrieți o variantă generalizată a clasei `Buffer`, numită `MessageQueue`, care permite în plus comunicarea între producători și consumatori pe anumite “canale” independente. Aceasta înseamnă că clasa `MessageQueue` va putea fi utilizată drept suport de comunicare între categorii distincte de fire de execuție. De exemplu, un canal al clasei `MessageQueue` (să-l numim `LogChannel`) va putea fi folosit de către fire de execuție (producători) care doresc să scrie mesaje de log. Pot exista mai mulți consumatori care sunt înscrși la acest canal și preiau mesajele din coadă pentru a le prelucra într-un anumit mod. De exemplu, un consumator poate scrie mesajele de log într-o fereastră utilizator, în timp ce altul le va putea scrie într-un fișier text. Mai precis, clasa `MessageQueue` va trebui să implementeze interfața `IMessageQueue` de mai jos:

```

1 public interface IMessageQueue
2 {
3     /** Creeza un canal nou cu numele name.
4      * Se initializeaza o coada noua de
5      * asteptare pentru a stoca mesajele
6      * care vin pe acest canal.
7      */
8     public Channel createChannel( String name ) ;
9
10    /** Se apeleaza pentru a inchide un canal */
11    public void destroyChannel( Channel channel ) ;
12
13    /** Asteapta aparitia urmatorului mesaj, dupa care
14     * intoarce mesajul respectiv.
15     */
16    public Object getMessage( Channel channel ) ;
17
18    /** Adauga un mesaj in coada pentru un anumit canal. */
19    public void putMessage( Channel channel, Object message ) ;
20 }

```

Anexe

A. Mediul Java și uneltele ajutătoare

Geniul nu se mulțumește să
constate. El oferă soluții.

Robert Graves

Scopul acestei anexe este de a veni în ajutorul celor care încep să programeze în Java și a le indica cele mai adecvate unelte, necesare pentru a începe dezvoltarea de aplicații Java. În cadrul acestei anexe vom prezenta:

- Cele mai populare editoare și medii integrate Java;
- Cum se instalează Java pe Windows și Linux;
- Cum se pot compila automat aplicațiile Java folosind `ant` ;
- Site-uri web care conțin informații utile oricărui dezvoltator Java.

A.1 Editarea codului sursă

Pentru editarea codului sursă al unui program Java este suficient un simplu editor de fișiere text, cum ar fi *Notepad* (pe Windows) sau *pico* (pe Linux). Totuși, majoritatea programatorilor doresc facilități superioare celor pe care le poate oferi un astfel de editor de fișiere text. Există două variante în această situație: utilizarea unor editoare mai performante (paragraful A.1.1) sau utilizarea unor medii integrate de dezvoltare a aplicațiilor Java (paragraful A.1.2).

A.1.1 Editoare

Cea mai importantă facilitate oferită de următoarele editoare este "syntax highlighting" (colorarea textului în funcție de ceea ce reprezintă - de exemplu un

comentariu, numele unei clase, un cuvânt rezervat etc.). De asemenea, primele două editoare menționate în tabelul de mai jos oferă facilități suplimentare cum ar fi compilarea și execuția codului, organizarea fișierelor în proiecte, interfață pentru ant (vezi paragraful A.3.1) etc. Puteți afla mai multe despre fiecare editor în parte vizitând situl asociat, furnizat de tabelul de mai jos.

Editor	URL	OS/JDK	Observații
jEdit	jedit.org	J2SDK 1.3 și 1.4	Este scris în Java și rulează pe orice sistem care dispune de Java 2 versiunile 1.3 sau 1.4 (recomandat)
Jext	jext.org	J2SDK 1.3	Versiunea pentru Windows include un mediu Java
NEdit	nedit.org	Linux	Este scris în C și este foarte rapid

A.1.2 Medii integrate IDE (Integrated Development Environments)

Mediile integrate IDE oferă mult mai multe facilități decât un simplu editor. Practic, folosind un mediu IDE, programatorului îi sunt puse la dispoziție toate elementele de care are nevoie pentru a crea aplicații Java profesionale: *debugger*-e pentru executarea pas cu pas a aplicațiilor, modalități de a vedea în orice moment ierarhia de clase și metode existente în cadrul unui proiect, completarea automată a codului (code completion), compilare incrementală (codul este compilat în timp ce îl scrieți) etc. Dintre toate aceste produse, noi recomandăm cu căldură mediul Eclipse care este ușor de utilizat și dispune de o veritabilă armată de facilități deosebit de utile.

Aplicație	URL	OS/JDK	Observații
Eclipse	eclipse.org	J2SDK 1.3	Inclus și în produsele IBM VisualAge
Forte for Java (Sun ONE)	forte.sun.com/ffj	J2SDK 1.3	Bazat pe NetBeans
JCreator	jcreator.com	Windows	Scriș în C, este foarte rapid
JBuilder	borland.com/jbuilder/personal	Windows, Linux	Varianta "Personal Edition" este disponibilă gratuit.
NetBeans IDE	netbeans.org/ide	J2SDK 1.3	
VisualAge for Java	ibm.com/software/ad/vajava	Windows	Cea mai nouă versiune este 4.0

A.2 Instalarea mediului Java

Pentru a dezvolta și a rula programe Java, trebuie să instalați mediul de programare Java aflat în *kit*-ul de instalare al Java 2 SDK. Ultima versiune de Java disponibilă la momentul scrierii acestei lucrări este 1.4, iar kitul este disponibil gratuit la adresa <http://java.sun.com/j2se/1.4/download.html>. Fiecare sistem de operare are propriul lui kit de instalare, care va fi descărcat separat de la adresa anterioară. Sun oferă kit-uri de instalare pentru următoarele platforme:

- Windows;
- Linux;
- Solaris.

După ce descărcați kitul de instalare specific sistemului dumneavoastră de operare, puteți începe instalarea mediului Java, ajutându-vă la nevoie de următoarele două secțiuni ce descriu instalarea Java 2 SDK v1.4.0 (cea mai recentă versiune testată) pe două sisteme de operare frecvent folosite: Windows și Linux.

A.2.1 Instalarea sub Windows

Java 2 SDK pentru Windows poate fi utilizat sub versiunile Windows 95, Windows 98 (prima și a doua ediție), Windows NT 4.0 cu ServicePack 5, Windows ME, Windows XP Home, Windows XP Professional, Windows 2000 Professional sau Windows 2000 Advanced Server, pe tehnologie Intel. Pentru a putea rula cu succes Java este necesară o configurație minimală constând în procesor Pentium cu frecvența de 166MHz, 32 MB¹ RAM și 70MB spațiu liber pe hard-disk .

A.2.1.1 Instrucțiuni de instalare

În această secțiune, vom arăta cum se rulează programul de instalare pentru a dezarhiva și instala pachetul Java 2 SDK. Este posibil ca după ce Java 2 SDK a fost instalat să vi se ceară repornirea sistemului. Instalarea se face în următorii pași:

1. În cazul în care ați descărcat kitul de instalare de pe Internet, verificați dacă fișierul descărcat este complet. Fișierul `j2sdk-1_4_0-win.exe` trebuie să aibe dimensiunea de 37067134 octeți (*bytes*) pentru ca descărcarea kitului de instalare să fie considerată completă²;
2. În cazul în care aveți deja instalată o versiune *Beta* sau versiunea *Release Candidate*³ de Java 2 SDK 1.4, dezinstalați-o. Folosiți pentru aceasta funcția *Add/Remove Programs* din Windows, accesibilă din *Control Panel* (*Start -> Settings -> Control Panel*);
3. Fișierul `j2sdk-1_4_0-win.exe` reprezintă programul de instalare al Java 2 SDK. Rulați fișierul și urmăriți instrucțiunile simple care vă sunt furnizate de program;
4. Setați variabila sistem PATH. Setarea variabilei sistem PATH vă permite să puteți rula convenabil executabilele Java 2 SDK (`javac.exe`, `java.exe`, `javadoc.exe` etc.) din orice director fără să fie nevoie să scrieți întreaga cale în linia de comandă. Dacă nu setați variabila PATH, va trebui să specificați întreaga cale către executabil de fiecare dată când doriți să-l rulați, ca în exemplul de mai jos:

¹ megabytes

² Atenție totuși la versiunea kitului. Dacă ați ales o altă versiune a kitului Java 2 SDK (de exemplu, 1.3.0 etc.), mai mult ca sigur că dimensiunea fișierului va fi alta.

³ versiune anterioară versiunii finale


```
c:>\jdk1.4\bin\javac MyClass.java
```

Pentru a evita o asemenea sintaxă incomodă, adăugați întreaga cale a directorului `jdk1.4\bin` la variabila sistem `PATH`. Dacă la instalare ați ales opțiunile implicite, calea absolută în care se află executabilele Java este `c:\jdk1.4\bin`.

Setarea căii se realizează diferit, în funcție de versiunea Windows pe care o folosiți:

- pentru Windows NT, Windows 2000 și Windows XP: apăsați *Start, Settings, Control Panel*, dublu click pe *System*. În cazul Windows NT, selectați *tab-ul Environment*; la Windows 2000 selectați *tab-ul Advanced* și *Environment Variables*. Apoi, căutați variabila sistem `PATH` în *User Variables* sau *System Variables*. Dacă nu sunteți sigur unde anume doriți să adăugați calea către executabilele Java, adăugați-o la finalul variabilei `PATH` din *User Variables*. Configurarea căii conține calea absolută a directorului în care este instalat Java (de exemplu, `c:\jdk1.4\bin`). Scrierea cu litere mari sau mici nu are vreo influență. În final, apăsați *Set, OK* sau *Apply*. Variabila sistem `PATH` este o succesiune de directoare separate de ";". Sistemul de operare Windows caută executabilele Java (`javac`, `java`, `javadoc`, etc.) în directoarele menționate în cale, parcurgând variabila `PATH` de la stânga la dreapta. Este important de reținut că în `PATH` trebuie să aveți un singur director `bin` pentru Java SDK la un moment dat (celelalte directoare de acest fel care urmează după primul, sunt ignorate). Noua cale are efect în fiecare nouă fereastră *Command Prompt* pe care o deschideți sau în fiecare program pe care îl executați de acum înainte (cum ar fi JEdit, JCreator etc.).

- pentru Windows 98, Windows 95: editați fișierul `autoexec.bat` (folosind de exemplu *Start -> Run sysedit*). Căutați secvența în care este definită variabila `PATH` și adăugați în final calea absolută către executabilele Java. Dacă nu există nici o secvență de definire a variabilei `PATH`, adăugați-o pe o poziție oarecare în fișier. Iată un exemplu tipic pentru setarea variabilei `PATH`:

```
PATH C:\WINDOWS;C:\WINDOWS\COMMAND;c:\jdk1.4\bin
```

Scrierea cu literele mari sau mici nu are nici o influență. Pentru ca setarea căii să devină funcțională, executați următoarea instrucțiune într-o fereastră *Command Prompt*

```
c:\> c:\autoexec.bat
```

Pentru a afla setarea curentă a căii sau pentru a vedea dacă a avut efect schimbarea, tastați în linia de comandă:

```
c:\> path
```

În acest moment, sistemul dumneavoastră este pregătit pentru utilizarea Java 2 SDK. Pentru a verifica acest lucru, executați într-un *Command Prompt* comanda:

```
c:\>java -version
```

Dacă instalarea a decurs fără probleme, va fi afișat un scurt mesaj cu date despre versiunea Java instalată. Dacă primiți un răspuns similar cu:

```
The name specified is not recognized as an internal  
or external command, operable program or batch file.
```

înseamnă că mediul Java nu a fost corect instalat sau variabila PATH nu are o valoare corectă.

Pentru dezinstalarea Java 2 SDK, folosiți funcția *Add/Remove Programs* din *Control Panel*.

A.2.2 Instalarea sub Linux

Java 2 SDK rulează de platformele Intel Pentium sau compatibile cu kernel Linux v2.2.12 și glibc v2.1.2-11 sau versiuni mai recente. Aveți nevoie de minim 32 MB RAM de memorie și un spațiu liber pe hard-disk de 75MB. Puteți verifica versiunea curentă de glibc folosind comanda:

```
ls /lib/libc-*
```

A.2.2.1 Instrucțiuni de instalare

Pachetul Java 2 SDK, Standard Edition, v1.4.0 este disponibil în două formate de instalare:

- Un fișier binar `j2sdk-1_4_0-linux-i386.bin` care se autoextrage (autodezarhivează) și astfel Java 2 SDK se poate instala în orice locație de pe disc. Pentru această metodă, consultați secțiunea A.2.2.2;
- Fișierul `j2sdk-1_4_0-linux-i386-rpm.bin` care conține pachete tip RPM (Red Hat Package Manager) cu Java 2 SDK. Pentru a instala folosind această metodă, consultați secțiunea A.2.2.3.

Fișierele pentru ambele formate de instalare sunt cuprinse în cadrul unui fișier *shell script*.bin care afișează licența înainte de a începe instalarea.

A.2.2.2 Instalare cu autoextragere

Pentru instalare, executați pașii următori:

1. Verificați dimensiunea fișierului `j2sdk-1_4_0-linux-i386.bin`. O dimensiune de 40618207 bytes indică o descărcare corectă a kitului de instalare. Dacă mărimea fișierului nu corespunde celei de mai sus, înseamnă că fișierul a fost deteriorat în timpul descărcării. Singura alternativă este de a încerca din nou descărcarea fișierului;
2. Copiați `j2sdk-1_4_0-linux-i386.bin` în directorul în care doriți să instalați Java 2 SDK;
3. Rulați `j2sdk-1_4_0-linux-i386.bin` folosind comenzile:

```
chmod a+x j2sdk-1_4_0-linux-i386.bin
./j2sdk-1_4_0-linux-i386.bin
```

Scriptul de instalare va afișa o licență de utilizare, după care, dacă acceptați condițiile expuse în licență, Java 2 SDK se instalează în directorul curent;

4. Adăugați calea absolută către directorul bin rezultat în urma instalării (de exemplu, `/usr/local/jdk1.4/bin`) variabilei sistem PATH. Acest lucru se poate face astfel, funcție de *shell*-ul pe care îl utilizați:

- pentru *csh* sau *tcsh*:

```
% set PATH=( $PATH/usr/local/jdk1.4/bin )
```

- pentru *sh*:

```
% PATH=( $PATH/usr/local/jdk1.4/bin ); export $PATH
```

Puteți să adăugați liniile anterioare la sfârșitul fișierelor `.profile` sau `.cshrc` pentru a nu le mai scrie la fiecare *login*-are.

Pentru a verifica dacă instalarea a decurs fără probleme, deschideți o consolă și tastați comanda:

```
java -version
```

Dacă nu au fost semnalate erori pe parcursul procesului, atunci veți vedea un mesaj cu detalii despre versiunea Java instalată. Dacă primiți răspunsul:

```
java: Command not found
```

sau ceva similar, atunci mediul Java nu a fost bine instalat sau variabila PATH nu are o valoare corectă.

A.2.2.3 Instalarea folosind RPM

1. Verificați fișierul `j2sdk-1_4_0-linux-i386-rpm.bin`. O dimensiune de 39482030 bytes indică o descărcare corectă a kitului de instalare. Dacă mărimea fișierului nu corespunde celei de mai sus, înseamnă că fișierul a fost deteriorat în timpul descărcării. Încercați din nou să descărcați fișierul respectiv;

2. Rulați `j2sdk-1_4_0-linux-i386-rpm.bin` folosind comenzile:

```
chmod a+x j2sdk-1_4_0-linux-i386-rpm.bin
./j2sdk-1_4_0-linux-i386-rpm.bin
```

Scriptul va afișa o licență de utilizare, după care, dacă acceptați cerințele licenței, veți obține ca rezultat `j2sdk-1_4_0-linux-i386.rpm`, fișier ce va fi creat în directorul curent;

3. Accesați sistemul ca *root* folosind comanda `su` și parola acestui cont:

```
su - root
```

4. Dezinstalați J2SDK 1.4.0 Beta (în cazul în care există această versiune anterioară deja instalată pe calculatorul dumneavoastră)

Notă: Calea implicită pentru instalarea RPM a lui J2SDK este următoarea: `/usr/java/j2sdk1.4.0`. Acesta este directorul în care au fost instalate și *pre-release*⁴-urile versiunii 1.4 (în cazul în care acestea au fost instalate). Pentru a pregăti instalarea versiunii finale, în acest director dezinstalați orice versiune *pre-release* 1.4 care ar putea exista pe sistem. Dacă nu sunteți sigur că aveți o versiune *pre-release* 1.4, verificați acest lucru folosind comanda:

```
rpm -query -a | grep j2sdk-1.4.0
```

⁴versiune anterioară versiunii finale

Comanda va afișa versiunea instalată, cu numele pachetului RPM al acesteia. După ce ați aflat numele pachetului îl puteți dezinstala cu una din următoarele comenzi (în funcție de versiunea *beta* instalată în momentul curent):

- Pentru a dezinstala versiunea *Beta*:

```
rpm -e j2sdk-1.4.0-beta
```

- Pentru a dezinstala pachetul *Beta 2*:

```
rpm -e j2sdk-1.4.0-beta2
```

- Pentru a dezinstala pachetul *Beta 3*:

```
rpm -e j2sdk-1.4.0-beta3
```

5. Rulați comanda `rpm` pentru a instala pachetul Java 2 SDK v1.4.0:

```
rpm -iv j2sdk-1_4_0-linux-i386.rpm
```

Această comandă va instala Java 2 SDK pe calculatorul dumneavoastră;

6. Adăugați directorul `bin` rezultat în urma instalării la variabila sistem `PATH` (vezi pasul 4 de la instalarea cu autoextragere);

7. Ieșiți din contul *root*.

Puteți verifica dacă instalarea a fost încununată de succes în același mod ca și la instalarea cu autoextragere.

A.3 Compilarea codului sursă

Odată ce mediul de programare Java a fost instalat și prima aplicație Java a fost creată cu ajutorul unui editor sau mediu IDE, ne punem problema compilării acestui prim program.

Pentru a compila o aplicație Java nu este suficient un simplu compilator. Pentru ca o aplicație să poată fi compilată, compilatorul are nevoie de bibliotecile Java standard și eventualele biblioteci folosite de aplicație.

Compilatoarele sunt în general disponibile împreună cu o mașină virtuală Java, care dispune de bibliotecile Java standard și alte utilitare. Toate acestea

împreună formează un SDK (Standard Development Kit). Tabelul de mai jos prezintă câteva SDK-uri mai uzuale, cel mai des folosite fiind cele oferite de Sun și IBM.

JDK	URL	OS
Blackdown v1.3	blackdown.org	Linux
IBM Developer Kit for Windows, v1.3.0	ibm.com/developerworks/java/jdk	Windows
IBM Developer Kit for Linux, Java 2 Technology Edition, v1.3	ibm.com/developerworks/java/jdk	Linux
Sun Java 2 Platform, Standard Edition, v1.4	java.sun.com/j2se/1.4	Windows, Linux, Solaris

Compilatoarele incluse într-un SDK se apelează cu comanda `javac` (sau comanda `jikes` în cazul compilatorului realizat de IBM).

Pentru a compila un program Java (să presupunem că am creat un program numit `PrimulProgram.java`), trebuie să executăm într-un *Command Prompt* (consolă), în directorul în care este salvat fișierul `.java`, comanda următoare:

```
javac PrimulProgram.java
```

Dacă în urma executării acestei comenzi nu s-a afișat nici o eroare și a fost creat un fișier `PrimulProgram.class`, atunci programul nu are greșeli de sintaxă și putem să îl executăm.

Pentru ca bibliotecile necesare compilării unei aplicații să fie găsite de către compilator, acestea trebuie specificate în variabila sistem `CLASSPATH`. `CLASSPATH` (asemănător cu variabila sistem `PATH`) reprezintă o listă de directoare, arhive ZIP și fișiere în format JAR ("jar"-urile sunt de fapt arhive standard ZIP). Obiectele listei sunt despărțite prin caracterul `';` pe Windows și prin caracterul `':'` pe Linux. De exemplu, pe Windows 98 variabila `CLASSPATH` se poate seta din `autoexec.bat` introducând comanda (evident, numele de directoare sunt prezentate orientativ):

```
set CLASSPATH=d:\api\interfata.jar;d:\api\upload\;.
```

iar pe Linux aceasta se poate seta din `/etc/profile` cu comanda:

```
export CLASSPATH=/usr/local/jars/xerces.jar:
/root/work/server_upload/;.
```

Setarea CLASSPATH se realizează într-un mod analog cu cel al variabilei PATH (vezi secțiunea A.2.1.1).

Compilatorul Jikes, realizat de IBM, este disponibil gratuit la următoarea adresă: www.alphaworks.ibm.com/formula/Jikes/. Acesta nu include bibliotecile standard și se folosește de obicei împreună cu SDK-ul de la Sun, înlocuind compilatorul standard `javac`, căruia îi este superior ca viteză de compilare și descriere a erorilor. Pentru a putea rula cu succes `jikes` trebuie ca bibliotecile standard oferite de mediul Java (aflate în `jre/lib/rt.jar`) să fie incluse în variabila sistem CLASSPATH.

A.3.1 Ant

Ant este o aplicație gen “*make*” (familiară utilizatorilor de Linux) care permite automatizarea diferitelor operații, cum ar fi compilarea unei aplicații și pregătirea ei pentru a fi distribuită sau livrată. *Ant* este foarte util în cazul aplicațiilor Java complexe, care necesită multe operații pentru a fi transformate din cod sursă în cod binar gata de a fi rulat. De exemplu, pentru a distribui o aplicație Java pentru telefoane mobile sau PDA-uri sunt necesare următoarele operații:

- compilarea fișierelor sursă;
- preverificarea fișierelor compilate;
- crearea unei arhive `jar` care să conțină fișierele sursă compilate, eventuale resurse (imagini, sunete etc.) și alte biblioteci;
- obfuscarea⁵ arhivei rezultate;
- crearea unui descriptor al aplicației (fișier `JAD`);
- conversia arhivei într-un format acceptat de telefonul mobil sau PDA-ul pe care se va rula.

Cititorii care nu sunt familiarizați cu dezvoltarea de aplicații pe unități mobile nu au înțeles probabil mare lucru din pașii de mai sus. Am vrut doar să punem în evidență faptul că de la scrierea codului pentru o aplicație până la execuția codului pe platforma destinație poate fi cale lungă, iar realizarea manuală a operațiilor necesare poate fi extrem de anevoioasă. Există multe soluții care

⁵Obfuscarea este un proces prin care se reduce dimensiunea byte-codului, folosind diverse optimizări (de exemplu, eliminarea metodelor și atributelor care nu sunt folosite, înlocuirea getter-ilor și setter-ilor cu atributele asupra cărora acționează etc.).

automatizează acest proces, iar unul care se distinge prin eleganță și aria extrem de largă de aplicabilitate este Ant. Practic, pentru exemplul anterior, compilarea și pregătirea aplicației spre distribuție devine o simplă rulare a comenzii ant.

La execuția comenzii ant, se caută mai întâi în directorul curent fișierul cu numele build.xml. Acest fișier trebuie creat manual; el conține mai multe *scopuri* (engl. targets) care asigură îndeplinirea anumitor sarcini. Lansând ant fără nici un parametru se execută scopul implicit, care de obicei este cel care asigură compilarea fișierelor sursă ale aplicației. Pentru a executa alt scop, de exemplu scopul cu numele "build", se execută comanda

```
ant build
```

Iată un exemplu de fișier build.xml:

```
<project name="MyComplexApp" default="compile"
    basedir=".">

<target name="compile">
    <copy todir="bin">
        <fileset dir="." includes="*.properties"/>
    </copy>
    <javac srcdir="src" destdir="bin" debug="on"
        optimize="on"/>
</target>

<target name="build" depends="compile">
    <jar jarfile="MyComplexApp.jar" basedir="bin">
</target>

</project>
```

În acest exemplu, scopul compile copiază fișierele .properties (de exemplu jndi.properties) în directorul bin și apoi compilează toate clasele din directorul src în directorul bin, păstrând structura de directoare. Scopul build depinde de scopul compile, deci când se lansează ant build, mai întâi se execută scopul compile și dacă nu apare nici o eroare se execută scopul build. Scopul build arhivează fișierele din directorul bin într-o arhivă de tip jar numită MyComplexApp.jar.

Prezentarea Ant din cadrul acestui paragraf este mai mult decât sumară, rolul ei fiind doar de a vă deschide apetitul pentru procesarea automată. Mai multe detalii puteți afla la adresa <http://jakarta.apache.org/ant>.

A.4 Rularea unei aplicații java

Execuția unei aplicații Java implică existența (pe lângă bibliotecile folosite de aplicație) unei mașini virtuale Java și a bibliotecilor standard (care conțin clasele `java.lang.*`, `java.util.*`, `java.io.*` etc.). Acestea sunt incluse în SDK-uri și JRE-uri (Java Runtime Environment). Pentru a rula o aplicație Java este suficient un JRE (mediu de execuție Java) care, spre deosebire de SDK, nu conține utilitățile pentru dezvoltarea programelor (de exemplu, `javac`), ci doar pe cele necesare execuției lor.

Este necesar să clarificăm niște denumiri. Până la apariția versiunii 1.2 a JDK-ului (Java Development Kit), denumirea Java 2 Platform Standard Edition (J2SE) nu a existat. Începând cu acea versiune, a apărut Java 2 Platform Standard Edition cu SDK 1.2 (Standard Development Kit). În continuare au apărut SDK 1.3 și SDK 1.4 care țin tot de Java 2 Platform Standard Edition.

Revenind, pentru a rula de exemplu `PrimulProgram.class` se folosește comanda următoare, executată în directorul curent:

```
java PrimulProgram
```

A.4.1 CLASSPATH

Bibliotecile necesare rulării unei aplicații (cu excepția bibliotecilor standard) trebuie incluse în variabila sistem `CLASSPATH` (așa cum am descris în paragraful A.2) sau pot fi specificate în linia de comandă, folosind opțiunea `-classpath <CLASSPATH>`.

Utilitățile din SDK folosesc următoarea ordine pentru a căuta o clasă:

- clasele din `<JAVA_HOME>/jre/lib/rt.jar`, urmate apoi de clasele din arhiva `<JAVA_HOME>/jre/lib/i18n.jar`;
- clasele din `<JAVA_HOME>/jre/lib/ext/*.jar`;
- clasele din variabila `CLASSPATH`.

De exemplu dacă este căutată clasa `MyClass` din pachetul `com.mypackage` și variabila `CLASSPATH` are valoarea `/root/work/server_upload:/usr/local/jars/xerces.jar`, atunci fișierul `MyClass.class` va fi căutat astfel:

1. în arhivele din `jre/lib`;
2. în directorul `/root/work/server_upload/com/mypackage/`;

3. în arhiva `/usr/local/jars/xerces.jar`, subdirectorul `com/my-package/`.

A.5 Documentații java

Principala sursă de documentare pentru limbajul Java este *site*-ul oficial al companiei Sun Microsystems: `java.sun.com`.

Documentația Java 2 Platform Standard Edition se găsește la adresa de Internet: `http://java.sun.com/j2se/1.4/docs`. Acolo poate fi găsită și funcționalitatea bibliotecilor standard (API).

Pentru a genera documentații în format HTML din fișiere sursă Java, se folosește utilitarul `javadoc`.

B. Convenții de notație în Java. Utilitarul javadoc

În Mănăstire existau anumite reguli, dar Maestrul ne-a atras întotdeauna atenția asupra tiraniei regulilor.

Anthony de Mello, O clipă de înțelepciune.

Începutul înțelepciunii constă în a desemna obiectele cu numele lor adecvate.

Proverb chinezesc
Femeile și pisicile vor face
întotdeauna ceea ce vor, iar
bărbații și câinii trebuie să se
relaxeze și să se împace cu ideea.

Robert A. Heinlein

B.1 Necesitatea convențiilor de scriere a programelor

Convențiile de scriere sunt foarte importante pentru programatori din mai multe motive:

- Conform studiilor de specialitate, 80% din costul unei aplicații software este datorat întreținerii aplicației respective;

- Este foarte puțin probabil ca aplicația să fie întreținută pe întreg parcursul existenței ei doar de către cei care au dezvoltat-o;
- Convențiile de notație îmbunătățesc lizibilitatea codului, permițând programatorilor să înțeleagă codul nou mai rapid și mai profund;
- Dacă aplicația scrisă de dumneavoastră este distribuită ca produs, trebuie să vă asigurați că este bine realizată și organizată.

Există deci motive foarte întemeiate pentru a utiliza convențiile de scriere a programelor. Probabil că cititorii vor observa faptul că uneori aceste convenții nu au fost respectate pe parcursul lucrării. Situația este explicabilă prin faptul că spațiul disponibil a fost limitat, ceea ce a condus la evitarea utilizării convențiilor mai "costisitoare" din punct de vedere al spațiului tipografic (de exemplu, abundența de comentarii `javadoc`). Totuși, sugestia noastră este să nu evitați folosirea acestor convenții în programele dumneavoastră, indiferent de mărimea și complexitatea aplicațiilor.

B.2 Tipuri de fișiere

Java folosește următoarele extensii pentru fișiere:

- `.java`, pentru fișiere sursă;
- `.class`, pentru fișiere compilate (*bytecode*).

Dacă există mai multe fișiere într-un director, atunci este benefică prezența unui fișier `README`, care să descrie conținutul directorului.

B.3 Organizarea fișierelor sursă `.java`

Un fișier sursă este împărțit pe secțiuni, separate prin linii goale și, opțional, un comentariu ce identifică fiecare secțiune. Fișierele cu peste 2.000 de linii sunt incomod de parcurs și modificat și trebuie pe cât posibil evitate. Vom prezenta un exemplu elocvent de program Java formatat în mod corespunzător în paragraful B.10.

Fiecare fișier sursă Java conține o singură clasă publică sau interfață. Când clasa publică este asociată cu alte clase sau interfețe ne-publice acestea se pot scrie în același fișier sursă cu clasa publică, dar clasa publică trebuie să fie *prima* în cadrul fișierului.

Secțiunile unui fișier sursă Java sunt:

- comentariile de început;
- instrucțiunile `package` și `import`;
- declarațiile clasei și/sau interfeței.

B.3.1 Comentariile de început

Conțin informații legate de *copyright* și o descriere a fișierului.

Informația *copyright* are următorul format standard:

```
1 /*
2  * COPYRIGHT NOTICE
3  * This file contains proprietary information of ...
4  * Copying or reproduction without prior written
5  * approval is prohibited.
6  * Copyright (c) xxxxx
7  */
```

Descrierea fișierului conține observații legate de numele fișierului, funcționalitatea lui, modul de utilizare a claselor sau interfețelor care îl compun și alte informații adiționale. Comentariul trebuie să fie în stil javadoc (informații detaliate despre utilitarul javadoc în paragraful B.11):

```
1 /**
2  * File name: Salut.java
3  * Description : program ce afiseaza un salut de bun venit
4  *               in lumea Java. Contine clasele: ...
5  * Usage : standard (java Salut)
6  * Version : 1.0
7  * Date : 15 ianuarie 2002
8  */
```

B.3.2 Instrucțiunile `package` și `import`

Numele pachetului trebuie să apară pe prima linie necomentată din fișierul sursă și trebuie să respecte convenția de notare stabilită în această anexă. Imediat după aceasta, trebuie să apară declarațiile de `import` ale claselor. De exemplu:

```
1 package test.capitolul4;
2
3 import java.awt.Graphics;
4 import java.util.*;
```

B.3.3 Declarațiile clasei și interfeței

O clasă/interfață trebuie să fie declarată astfel (exact în această ordine):

1. comentariul de tip javadoc (`/** */`), în care se precizează informații publice despre clasă/interfață;
2. cuvântul cheie `class` sau `interface`, urmat de numele clasei;
3. opțional, comentariu legat de implementarea clasei/interfeței (`/** */`). Conține informații private, neneesare pentru utilitarul javadoc;
4. atributele `static`. Mai întâi cele `public`, apoi cele `protected`, cele fără modificatori de acces și, în final, cele `private`. Evident, în categoria atributelor statice intră și constantele;
5. atributele nestatice, în aceeași precedență ca și cele statice;
6. constructorii clasei;
7. metodele clasei. Metodele trebuie grupate din punct de vedere al funcționalității, și nu din punct de vedere al accesibilității. De exemplu, o metodă statică privată s-ar putea afla între două metode nestatice publice. Scopul este de a face codul cât mai ușor de citit și înțeles. Fiecare metodă este precedată de o descriere precisă în format javadoc.

Documentarea este obligatorie pentru metodele publice. Metodele standard de acces la atribut (metodele de tipul `get/set`) pot fi grupate fără a avea o descriere, funcționalitatea lor fiind binecunoscută (modifică sau returnează valoarea unui atribut al clasei). Standardul stabilit pentru descrierea metodelor este realizat în așa fel încât descrierea să fie ușor de înțeles. Multe elemente sunt opționale și pot fi omise. Fiecare element de pe linie începe cu simbolul asterisk (*) și se termină cu spațiu. Dacă un element este prezentat pe mai multe linii, atunci fiecare linie începând cu a doua trebuie indentată, astfel încât fiecare linie să fie aliniată vertical cu cea precedentă.

Un exemplu arată astfel:

```
1 /**
2  * <descriere detaliata a metodei>
3  *
4  * @param      <descrierea fiecarui parametru al metodei>
5  * @return     <explicarea tipului de valoare returnat>
6  * @exception  <explicarea fiecărei excepții aruncate>
7  * @author     <numele autorului>
8  */
```

Cuvintele `@param`, `@return`, `@exception`, `@author` reprezintă etichete speciale recunoscute de javadoc (paragraful B.11).

Descrierea detaliată a metodei trebuie să cuprindă:

- rolul metodei;
- pre și post-condiții;
- efecte secundare;
- dependențe de alte metode/clase;
- ce altă metodă ar trebui să apeleze această metodă;
- dacă metoda trebuie sau nu să fie redefinită în clasele derivate.

Secțiunea `@param` descrie tipul, clasa, constrângerile tuturor argumentelor metodei. De exemplu:

```
* @param    strSursa - stringul de intrare.
                Nu poate fi de dimensiune 0.
```

Secțiunea `@return` descrie valoarea returnată de metodă: tipul de date returnat, domeniul de valori în care se poate afla valoarea returnată și, dacă este posibil, informația de eroare returnată de metodă. De exemplu:

```
* @return    returneaza un intreg din multimea 1..n.
```

Secțiunea excepțiilor oferă o descriere a excepțiilor pe care metoda le aruncă. De exemplu,

```
* @exception ResourceNotFoundException - daca resursa
                nu a fost gasita.
```

Deși este evident, adăugăm un exemplu și pentru secțiunea autor:

```
* @author    Mihai Ionescu
```

B.4 Indentarea

Codul sursă trebuie să fie indentat, folosind *tab*-uri și nu spații. Indentarea cu *tab*-uri are avantajul că în aproape toate editoarele de texte se poate modifica nivelul de indentare selectând dimensiunea preferată pentru caracterul *tab*. Aceasta este în general setată la două (2) spații, dar se consideră acceptabil și un *tab* de patru (4) spații.

B.4.1 Lungimea unei linii de cod

Trebuie evitate liniile de cod mai lungi de 80 de caractere, pentru că nu pot fi tratate corect de anumite terminale și editoare. De asemenea, liniile care alcătuiesc comentariul javadoc nu trebuie să depășească 70 de caractere.

B.4.2 "Ruperea" liniilor

În situația în care o anumită expresie nu este total vizibilă pe ecran (are o lungime prea mare), este indicată "ruperea" ei, pentru ca tot codul sursă să fie vizibil, fără să fie nevoie de *scroll*-area orizontală a fișierului. Regulile după care se realizează această operație sunt următoarele:

- "rupere" după o virgulă;
- "rupere" înaintea unui operator;
- alinierea liniei noi cu începutul expresiei la același nivel cu linia precedentă;
- dacă liniile de cod sunt confuze, indentați liniile noi cu 1-2 *tab*-uri.

De exemplu:

```
1 numeLung1 = numeLung2 * ( numeLung3 + numeLung4 - numeLung5 )
2           + 4 * numeLung6 ; //ASA DA
3
4 numeLung1 = numeLung2 * ( numeLung3 + numeLung4 -
5                       numeLung5 ) + 4 * numeLung6 ; //ASA NU
```

B.4.3 Acoladele

Acolada de deschidere trebuie să fie pe linia următoare, aliniată vertical cu linia precedentă. Acolada de închidere trebuie să fie pe o linie separată, aliniată vertical cu acolada de deschidere.

Exemplu:

```
1 if ( conditie )
2 {
3     ...
4 }
```

Este acceptată și deschiderea acoladei pe aceeași linie cu ultima instrucțiune, și închiderea aliniată vertical cu instrucțiunea care a deschis acolada:


```

1 if ( conditie ){
2     ...
3 }

```

Alegerea uneia dintre cele două variante este o chestiune de preferință personală. Prima are avantajul de a crește lizibilitatea, dar unii se plâng că ocupă prea mult spațiu. După cum ați observat studiind codul din această lucrare, autorii preferă să utilizeze prima variantă.

B.4.4 Spațierea

Liniile goale plasate adecvat pot îmbunătăți lizibilitatea codului, împărțind codul în secțiuni logice.

Două linii vide trebuie folosite în următoarele circumstanțe:

- între secțiunile unui fișier sursă;
- între definițiile claselor și interfețelor.

O linie vidă trebuie folosită în următoarele situații:

- între metode;
- între variabilele locale ale unei metode și prima sa instrucțiune;
- înaintea unui comentariu de tip bloc sau a unui comentariu pe o singură linie (vezi secțiunea 5.1 din prezenta anexă).

Spațiile trebuie folosite în următoarele situații:

- un cuvânt cheie urmat de o paranteză trebuie separat prin spațiu de aceasta. Exemplu:

```

1 while ( true )
2 {
3     ...
4 }

```

Trebuie reținut faptul că spațiul nu este interpretat ca separator de către compilatorul Java atunci când se află între numele unei metode și paranteza care deschide lista de argumente a metodei;

- un spațiu trebuie să apară după virgulă (,) în lista de argumente;

- toți operatorii binari, cu excepția punctului (.) trebuie separați de operandi prin spațiu. Spațiile nu se folosesc niciodată pentru a separa operatorii unari și cei de incrementare/decrementare (++/--). De exemplu:

```
1 a = ( b + c ) / ( d + e );  
2 b++;  
3 printSize( a , b , c );
```

- expresiile dintr-o instrucțiune `for` trebuie separate prin spațiu. De exemplu:

```
for ( expr1 ; expr2 ; expr3 )
```

- operatorul de `cast` trebuie tot timpul urmat de spațiu:

```
myMethod(( byte ) unNumar , ( Object ) x );
```

B.5 Comentariile

Programele Java pot avea două tipuri de comentarii: comentarii de implementare și comentarii pentru documentație. Comentariile de implementare sunt de tipul celor din C++, delimitate prin `/* . . . */` sau `//`. Comentariile pentru documentație, cunoscute sub numele de comentarii `javadoc`, sunt disponibile doar în Java și sunt delimitate de `/** . . . */`. Comentariile `javadoc` pot fi extrase și formate în fișiere HTML folosind aplicația utilitară `javadoc`.

Comentariile de implementare aduc clarificări asupra codului sursă și sunt adresate programatorilor care lucrează sau vor lucra la scrierea codului. Comentariile de documentație aduc clarificări asupra rolului fiecărei clase, interfețe, metode etc. și sunt adresate celor care doresc să utilizeze clasele scrise de noi, fără a fi interesați de detaliile de implementare (codul sursă).

Comentariile conțin doar informații relevante pentru citirea și înțelegerea programului. În general, este bine să se evite duplicarea informației care este prezentată într-o formă clară în interiorul codului. De asemenea, este bine să se evite comentarii care ar putea deveni neactuale pe măsură ce codul sursă evoluează. Pe de altă parte, comentariile nu trebuie să fie folosite în mod abuziv. Frecvența mare a comentariilor reflectă uneori o calitate slabă a codului. Dacă simțiți nevoia să comentați foarte des codul, este cazul să luați în considerare rescrierea lui pentru a-l face mai clar.

B.5.1 Comentarii de implementare

Programele pot avea patru stiluri de comentarii de implementare: în bloc, pe o singură linie, comentarii de tip *trailing* și comentarii de tip sfârșit de linie.

- Comentarii de tipul bloc

Comentariile de tipul bloc sunt folosite pentru a oferi descrieri ale fișierelor, metodelor, structurilor de date și algoritmilor. Ele pot fi folosite la începutul fiecărui fișier și înainte de fiecare metodă. Dacă sunt folosite în interiorul unei metode, comentariile de tipul bloc trebuie să fie indentate pe același nivel cu codul sursă pe care îl descriu.

Exemplu:

```
1 /*
2  * Acesta este un comentariu de tip bloc
3  */
```

- Comentarii pe o singură linie

Comentariile mai scurte pot să apară pe o singură linie, indentată la nivelul codului care urmează. Dacă nu poate fi scris pe o singură linie, comentariul trebuie transformat într-un comentariu de tip bloc.

Exemplu:

```
1 if ( conditie )
2 {
3     /* Trateaza cazul cand conditia este adevarata. */
4     ...
5 }
```

- Comentarii de tipul *trailing*

Comentariile foarte scurte pot să apară pe aceeași linie cu codul pe care îl descriu, dar trebuie să fie indentate mai mult pentru a le separa de cod. Toate comentariile de acest tip care apar într-o porțiune de cod trebuie să aibe aceeași indentare.

Exemplu:

```
1 if ( a == 2 )
2 {
3     return true;          /* caz special, 2 este prim */
4 }
5 else
6 {
7     return isPrime(a);    /* verifica daca a este prim */
8 }
```

- Comentarii de tipul sfârșit de linie

Delimitatorul `//` comentează o linie complet sau parțial.

Exemplu:

```
1 if (myAge > 1)
2 {
3     // executa o operatie
4     ...
5 }
```

B.5.2 Comentariile de documentație

Platforma Java oferă o aplicație utilitară, denumită `javadoc`, foarte utilizată în crearea de documentații pentru pachetele și clasele Java. Practic, această aplicație a impus un standard în materie de documentare a programelor. Comentariile de documentație descriu clasele, interfețele, constructorii, metodele și atributele. Fiecare astfel de comentariu se găsește între delimitatorii `/** ... */`, cu specificația că există câte un singur comentariu pentru fiecare clasă, interfață sau membru al clasei.

Exemplu de comentariu de documentație:

```
1 /**
2  * Clasa Vehicle ofera ...
3  */
4 public class Vehicle
5 {
6     ...
7 }
```

Comentariile de documentație nu trebuie poziționate în cadrul vreunei metode sau constructor, pentru că Java asociază aceste comentarii cu prima declarație de metodă pe care o întâlnește *după* comentariu. Amănunte despre modul de utilizare `javadoc` sunt prezentate în paragraful B.11.

B.6 Declarații

B.6.1 Numărul de declarații pe linie

Se recomandă folosirea unei singure declarații de variabilă pe o linie, deoarece astfel se încurajează comentarea liniilor respective. De exemplu, varianta

```
int i; // indicele sirului
int dim; // dimensiunea sirului
```

este preferată variantei

`int i, dim;`

B.6.2 Inițializarea variabilelor

Este indicat ca inițializarea variabilelor să se facă în momentul declarării. Există situații în care acest lucru nu este posibil, de exemplu când valoarea inițială a variabilei depinde de anumite calcule care trebuie mai întâi efectuate.

B.6.3 Locul de declarare a variabilelor

Cel mai bine este ca declarațiile variabilelor să fie realizate la începutul blocului în care sunt folosite (un bloc este delimitat de acolade { }), pentru a evita posibilele confuzii de notație.

B.6.4 Declararea claselor și a interfețelor

În cazul declarării de clase sau interfețe trebuie respectate următoarele reguli de formatare:

- nu există spațiu între numele metodei și paranteza " (" cu care începe lista de parametri;
- metodele sunt despărțite printr-o linie vidă

Exemplu:

```

1 class Test
2 {
3     void firstMethod ()
4     {
5         ...
6     }
7
8     void secondMethod ()
9     {
10        ...
11    }
12 }
```

B.7 Instrucțiuni

B.7.1 Instrucțiuni simple

Pentru claritate, fiecare linie trebuie să conțină cel mult o instrucțiune. Tendința generală a programatorilor începători de a scrie mai multe instrucțiuni pe o singură linie, pentru ca astfel programele să pară mai “mici” este puerilă și nejustificată. Un program aerisit, în care instrucțiunile se succed una sub cealaltă este mult mai ușor de parcurs și de înțeles decât un program îmbâcsit și înghesuit, chiar dacă acesta este de trei ori mai scurt:

```
1 argv++; //ASA DA
2 argc--; //ASA DA
3
4 argv++; argc--; //ASA NU
```

B.7.2 Instrucțiuni compuse

Instrucțiunile compuse reprezintă o listă de instrucțiuni cuprinse între acolade { }.

- instrucțiunile compuse se indentează cu un nivel față de instrucțiunea care le compune:

```
1 if (conditie)
2 {
3     instructiune_indentata_cu_un_nivel;
4 }
```

- acolada de deschidere trebuie să fie pe o linie nouă aliniată vertical cu linia precedentă, iar acolada de închidere trebuie să fie pe o linie nouă, aliniată vertical cu acolada de deschidere:

```
1 if (conditie)
2 { //aliniere verticala cu instructiunea
3
4 } //aliniere verticala cu acolada de deschidere
```

- acoladele trebuie folosite pentru toate instrucțiunile, chiar și pentru o singură instrucțiune, atunci când fac parte dintr-o structură de tipul `if-else` sau `for`. Astfel este mai simplu de adăugat instrucțiuni, fără a genera accidental erori datorate uitării introducerii de acolade, iar codul devine mai clar.

B.7.3 Instrucțiunea `return`

Instrucțiunea `return` utilizată împreună cu o valoare nu trebuie să folosească parantezele decât dacă acestea fac valoarea returnată mai evidentă într-un anumit fel. De exemplu:

```
1 return ;
2 return age ;
3 return (dim ? dim : defaultDim);
```

B.7.4 Instrucțiunea `if-else`

Instrucțiunea `if-else` trebuie să aibă următoarea formă:

```
1 if ( conditie )
2 {
3     instructiuni ;
4 }
```

sau:

```
1 if ( conditie )
2 {
3     instructiuni ;
4 }
5 else
6 {
7     instructiuni ;
8 }
```

sau:

```
1 if ( conditie )
2 {
3     instructiuni ;
4 }
5 else if ( conditie )
6 {
7     instructiuni ;
8 }
9 else
10 {
11     instructiuni ;
12 }
```

Instrucțiunea `if` folosește întotdeauna acoladele. Evitați următoarea formă generatoare de erori logice:

```
1 if ( conditie ) //EVITATI! SE OMIT ACOLADELE.
2
3     instructiune ;
```

B.7.5 Instrucțiunea `for`

O instrucțiune `for` are următorul format:

```
1 for ( initializare ; conditie ; actualizare )
2 {
3     instructiuni ;
4 }
```

O instrucțiune `for` vidă trebuie să fie de următoarea formă:

```
for ( initializare ; conditie ; actualizare )
;
```

B.7.6 Instrucțiunea `while`

O instrucțiune `while` trebuie să fie de următoarea formă:

```
1 while ( conditie )
2 {
3     instructiuni ;
4 }
```

O instrucțiune `while` vidă trebuie să fie de următoarea formă:

```
while ( conditie )
;
```

B.7.7 Instrucțiunea `do-while`

O instrucțiune `do-while` are următoarea formă:

```
1 do
2 {
3     instructiuni ;
4 }
5 while ( conditie );
```

B.7.8 Instrucțiunea `switch`

Instrucțiunea `switch` trebuie să arate astfel:

```
1 switch ( conditie )
2 {
3     case ABC:
4         instructiuni ;
5         /* NU are break */
6 }
```



```

7   case DEF:
8       instructiuni;
9       break;
10
11  case XYZ:
12      instructiuni;
13      break;
14
15  default:
16      instructiuni;
17      break;
18 }

```

De fiecare dată când un `case` nu are instrucțiune `break`, precizați acest lucru printr-un scurt comentariu. Orice instrucțiune `switch` trebuie să includă un caz `default`.

B.7.9 Instrucțiunea `try-catch`

Formatul unei instrucțiuni `try-catch` este următorul:

```

1 try
2 {
3     instructiuni;
4 }
5 catch (ClasaDeTipExceptie e)
6 {
7     instructiuni;
8 }

```

Numărul de instrucțiuni `catch` poate fi mai mare de 1. O instrucțiune de tipul `try-catch` poate să includă și `finally`, care se execută indiferent dacă blocul `try` s-a executat cu succes sau nu:

```

1 try
2 {
3     instructiuni;
4 }
5 catch (ClasaDeTipExceptie e)
6 {
7     instructiuni;
8 }
9 finally
10 {
11     instructiuni;
12 }

```

B.8 Convenții de notație

Convențiile de notație fac programele mai ușor de înțeles datorită faptului că sunt mai ușor de citit. De asemenea, ele dau informații despre metode, despre identificatori – de exemplu dacă este o constantă, un pachet sau o clasă – informații care pot fi folositoare în înțelegerea codului.

- Pachete

Pachetele trebuie să aibă nume unice, pentru a evita posibile erori de identificare. Numele pachetului este format numai din litere mici ale alfabetului. Evitați folosirea majusculilor. Trebuie reținut faptul că aplicațiile utilitare Java fac diferența între literele mici și cele mari ale alfabetului (sunt *case-sensitive*). Prefixul unui nume de pachet trebuie să fie unul dintre numele de domeniu `com`, `edu`, `gov`, `mil`, `net`, `org`, sau unul dintre codurile formate din două litere care identifică țara, în conformitate cu standardul ISO 3166. De exemplu: `ro` pentru România, `de` pentru Germania, `fr` pentru Franța etc. Următoarele componente ale numelui unui pachet diferă în funcție de convențiile fiecărei companii producătoare de software. Aceste convenții pot specifica departamentul, proiectul etc. Numărul lor poate fi oricât de mare, dar este de preferat ca structura să nu fie prea stufoasă.

Exemple de nume de pachete:

```
com.sun.eng
com.ibm.jdk
ro.firmaMea.proiectul2.util
ro.firmaMea.proiectul2.login
```

- Clase

Numele claselor începe întotdeauna cu majusculă și este format din unul sau mai multe substantive. Dacă sunt mai multe substantive, atunci prima literă a fiecărui substantiv este și ea majusculă. Încercați să păstrați aceste denumiri simple. Utilizați cuvintele întregi în defavoarea abrevierilor (cu excepția cazului în care abrevierea este mai des utilizată decât numele întreg, cum este cazul HTML sau URL)

Exemplu de nume de clase:

```
class Vehicle;
class SortedVector;
```

Numele claselor derivate poate fi construit, dacă este posibil, prin inserarea unui cuvânt înaintea numelui clasei de bază: din clasa `Chart` se pot deriva clasele `AreaChart`, `LineChart`;

- Interfețe

Numele interfețelor respectă condițiile impuse numelor de clase. Programatorii sunt încurajați să folosească prefixul `I` pentru a denumi o interfață, dar acest lucru nu este obligatoriu.

Exemplu de nume de interfețe:

```
interface IBook;
interface Disk;
```

- Metode

Numele metodelor reprezintă acțiuni, deci vor conține verbe. Numele metodelor începe cu literă mică, dar fiecare cuvânt, începând cu al doilea în cazul în care există, va începe cu majusculă.

Exemplu de nume de metode:

```
void cancelOrder();
void run();
void mergeSort();
```

Numele metodelor standard de acces la atributele unei clase (metodele de tipul `get/set`) trebuie să conțină prefixul `get` sau `set`, după cum este cazul.

Exemplu:

```
1 /**
2  * Returnează valoarea atributului name din clasa.
3  */
4  String getName();
5
6 /**
7  * Modifică valoarea atributului name, atribuindu-i
8  * noua valoare newName.
9  */
10 void setName(String newName);
```

În cazul atributelor de tip `boolean`, metodele de tip `get` trebuie să folosească prefixele `is`, `can` sau `has`.

De exemplu, varianta:

```
//returnează valoarea booleana student
boolean isStudent(); //ASA DA
```

este de preferat variantei:

```
boolean getStudent (); //ASA NU
```

- Variabile

Numele variabilelor începe cu literă mică, dar fiecare cuvânt, începând cu al doilea în cazul în care există, va începe cu majusculă. Numele de variabile nu trebuie să înceapă cu linie de subliniere (_), sau simbolul dolar (\$), deși acest lucru este permis de limbajul Java.

Numele variabilelor trebuie să fie scurte, dar cu înțeles. Numele de variabile formate dintr-o singură literă trebuie folosite doar pentru variabile temporare: *i*, *j*, *k*, *m*, *n* sunt folosite pentru variabile de tip întreg, în timp ce *c*, *d*, *e* sunt folosite pentru variabile de tip caracter.

Exemplu de nume de variabile:

```
int i;  
char c;  
float carWidth;
```

- Constantele

Numele constantelor este scris în totalitate cu majuscule. Dacă sunt mai multe cuvinte în componența unui nume de constantă, atunci acestea sunt separate prin linie de subliniere (_).

Exemplu de nume de constante:

```
static final int MIN_WIDTH = 1;  
static final int MAX_WIDTH = 4;  
static final int GET_THE_CPU = 1;
```

B.9 Practici utile în programare

B.9.1 Referirea atributelor și a metodelor statice

Evitați folosirea unui obiect pentru a accesa metodele și atributele statice ale unei clase. Este de preferat să folosiți numele clasei, punând astfel clar în evidență faptul că este vorba despre un membru static:

```
1 staticMethod (); //OK  
2 MyClass.staticMethod (); //OK  
3  
4 MyClass cl = new MyClass ();  
5 cl.staticMethod (); //EVITATI
```

B.9.2 Atribuirea de valori variabilelor

Evitați atribuirii multiple pe aceeași linie de cod, ca în exemplul:

```
Clasa1.car1 = Clasa2.car2 = 'c'; //ASA NU
```

Nu folosiți atribuirii imbricate pentru a îmbunătăți performanța execuției. Acest lucru va fi realizat de compilator:

```
1 d = ( a = b + c ) + r; //ASA NU
2
3 a = b + c; //ASA DA
4 d = a + r;
```

Folosiți parantezele în cadrul expresiilor cu mulți operatori, pentru a evita probleme de precedență a operatorilor:

```
1 if ( a == b && c == d ) //ASA NU
2
3 if (( a == b ) && ( c == d )) //ASA DA
```

Adăugați comentarii speciale anumitor părți de cod despre care știți că sunt susceptibile la erori sau nu funcționează deloc.

B.10 Exemplu de fișier sursă Java

Fișierul Java listat în continuare este un exemplu de program Java care respectă convențiile de programare prezentate în această anexă:

```
1
2 /*
3  * COPYRIGHT NOTICE
4  * This software is the confidential and proprietary
5  * information of MySoftware Ltd.
6  *
7  * Copyright (c) 2002 MySoftware Ltd.
8  * Silicon Alley 4B, Brasov, Romania
9  * All rights reserved.
10 */
11
12 package com.myPackage.vehicles;
13
14 import java.util.Vector;
15
16 /**
17  * FileName: Car.java
18  * Description: defineste o masina
19  *
20  * @version 1.00    15 Ianuarie 2002
```

```
21  * @author Ionescu Mihai
22  */
23  public class Car extends Vehicle
24  {
25      /* Comentariu de implementare a clasei. */
26
27      /** Comentariu de documentatie pentru classVar1. */
28      public static int classVar1;
29
30      /**
31       * Comentariul de documentatie pentru variabila classVar2
32       * este mai lung de o linie.
33       */
34      private static Object classVar2;
35
36      /** Comentariu de documentatie pentru instanceVar1. */
37      public Object instanceVar1;
38
39      /** Comentariu de documentatie pentru instanceVar2. */
40      protected int instanceVar2;
41
42      /** Comentariu de documentatie pentru instanceVar3. */
43      private Object [] instanceVar3;
44
45
46      /**
47       * Comentariu de documentatie pentru constructorul clasei
48       */
49      public Car()
50      {
51          // ... implementarea constructorului ...
52      }
53
54      /**
55       * Comentariu de documentatie pentru metoda doSomething.
56       */
57      public void doSomething()
58      {
59          // ... implementarea metodei ...
60      }
61
62      /**
63       * Comentariu de documentatie pentru metoda doSomethingElse.
64       * @param someParam descrierea parametrului
65       */
66      public void doSomethingElse(Object someParam)
67      {
68          // ... implementarea metodei ...
69      }
70 }
```

B.11 Utilitarul javadoc

Javadoc este o aplicație utilitară oferită de Sun Microsystems împreună cu JDK pentru a genera documentație API¹ în format HTML pornind de la comentariile de documentație din cadrul fișierelor sursă. Javadoc este inclus în distribuția J2SDK, sub forma unei aplicații cu același nume și se află în directorul bin al distribuției. Utilitarul nu poate fi obținut separat, ci doar ca parte integrantă a J2SDK.

javadoc este foarte bine documentat de către creatorii săi. Lucrarea de față va prezenta în cele ce urmează, doar informații absolut necesare pentru a putea folosi acest utilitar. Mult mai multe informații despre javadoc se pot găsi fie în cadrul documentației J2SDK, fie la adresa următoare:

<http://java.sun.com/j2se/javadoc/index.html>.

B.11.1 Sintaxa javadoc

Sintaxa generală javadoc arată astfel:

```
javadoc [optiuni] [nume-pachete] [fisiere-sursa]
        [@nume-fisiere]
```

Iată descrierea celor patru parametri opționali ai comenzii:

- **optiuni** reprezintă opțiunile utilitarului și vor fi detaliate pe parcursul acestei anexe;
- **nume-pachete** reprezintă numele pachetelor pentru care se crează documentația. Numele sunt separate prin spațiu, ca în exemplul `java.io java.util java.lang`. Fiecare pachet care va fi documentat trebuie specificat separat. De reținut faptul că javadoc folosește opțiunea numită `-sourcepath` pentru a căuta aceste pachete. Transmiterea ca argument a unui nume de pachet implică procesarea de către javadoc a tuturor fișierelor `.java` din directorul corespunzător pachetului respectiv, chiar și dacă fișierele `.java` reprezintă exemple de utilizare a anumitor clase sau, pur și simplu, sunt clase care nu fac parte din pachetul

¹ API = Application Program Interface. Acest termen desemnează metoda specifică prescrisă de sistemul de operare al unui calculator sau de o anumită aplicație prin care un programator care scrie o aplicație poate interacționa cu sistemul de operare sau aplicația respectivă. În cazul nostru, documentația API reprezintă o descriere în format HTML a claselor, interfețelor, metodelor, atributelor etc. din cadrul pachetelor Java, care are rolul de a descrie rolul acestora și modul în care pot fi utilizate de un programator.

respectiv. Acest lucru are loc din cauza faptului că javadoc nu parsează fișierele `.java` în căutarea unei declarații de pachet (package);

- `-fsiere-sursa` reprezintă numele fișierelor sursă, separate prin spațiu, fiecare dintre fișiere putând fi precedate de locația lor: `Button.java`, `awt/Graphics*.java`. Calea care precedă numele fișierului sursă determină unde va fi căutat acesta de javadoc. Javadoc *nu* folosește `-sourcepath` pentru a căuta fișierele sursă. Exemplul precedent arată totodată că este permisă folosirea de *wildcard*-uri (`Graphics*.java`). Acestea ușurează uneori foarte mult lucrul celui care realizează documentația, deoarece mai multe fișiere pot fi precizate printr-o singură comandă (de exemplu, `Graphics*.java` cuprinde denumirile tuturor fișierelor a căror denumire începe cu `Graphics`, este urmată de orice succesiune de caractere și au extensia `.java`: `Graphics.java`, `GraphicsStandard.java`, `GraphicsTemplate.java` etc.). Clasele sau interfețele pentru care se generează documentația completă prin rularea javadoc poartă numele de clase documentate;
- `@nume-fisiere` reprezintă fișiere care conțin numele de pachete sau de fișiere care vor fi documentate. Ordinea pachetelor și a claselor din cadrul fișierului nu are importanță, dar trebuie să apară câte una pe linie. Această opțiune este în special utilă pentru a evita apariția unor liste foarte "stufoase" de parametri reprezentând numele de pachete și fișierele sursă.

B.11.2 Descrierea utilitarului

Javadoc parcurge declarațiile și comentariile de documentație dintr-un set de fișiere Java și crează un set corespundent de pagini HTML, descriind (implicit) clasele `public` și `protected`, clasele interioare (*engl.* inner classes), interfețele, constructorii, metodele și atributele. Utilitarul poate fi folosit pe pachete întregi, pe fișiere sursă separate, dar și în variantă combinată, pe pachete și fișiere. În primul caz, argumentul utilitarului javadoc este o serie de nume de pachete, în timp ce în al doilea caz, argumentul este reprezentat de o serie de nume de fișiere sursă `.java`.

Pentru a putea fi utilizat, javadoc are nevoie de compilatorul Java. Javadoc utilizează `javac` pentru a compila declarațiile din fișierele sursă, ignorând implementarea membrilor claselor respective. În acest fel, javadoc construiește o bogată reprezentare internă a claselor, inclusiv ierarhia de clase și relațiile dintre clase, create prin utilizarea lor. Folosind această reprezentare, javadoc generează documentația HTML. Pe lângă aceste informații, javadoc

mai folosește și informațiile oferite de programator în cadrul comentariilor de documentație din fișierele sursă.

Când crează structura internă a claselor, `javadoc` încarcă toate clasele referite, cu alte cuvinte clasele sau interfețele care sunt explicit referite în definierea (implementarea) claselor/interfețelor documentate. Exemple de referire sunt tipul de valoare returnată, tipurile de parametri, tipul către care se realizează o operație de cast, clasa extinsă, interfața implementată, clasele importate, clasele folosite în definierea corpurilor metodelor, etc. Pentru a funcționa corect, `javadoc` trebuie să fie în măsură să găsească toate clasele referite. Faptul că `javadoc` utilizează compilatorul Java, `javac`, ne asigură de faptul că documentația HTML obținută corespunde exact cu implementarea propriu-zisă.

Conținutul și formatul documentației generate de `javadoc` pot fi modificate conform preferințelor folosind o componentă denumită *doclet*. Utilitarul `javadoc` are în mod implicit o astfel de componentă, denumită *doclet standard*, care generează documentația API în format HTML. Doclet-ul standard poate fi înlocuit cu altul care generează documentație în format PDF, XML, MIF, RTF sau alte formate. Există diverse doclet-uri create de Sun Microsystems, dar și de alte companii. Sun oferă trei doclet-uri: *doclet-ul standard*, prezentat anterior, *doclet-ul MIF*, care generează documentația API în format MIF (folosit de Adobe FrameMaker), PDF, PS sau RTF, și *doclet-ul DocCheck*, pentru verificarea corectitudinii comentariilor de documentație din fișierele sursă.

Lista doclet-urilor pe care le puteți folosi include și următoarele aplicații, realizate de diverse alte companii:

- *Dynamic Javadoc* (sau *DJavadoc*), ce generează HTML în format dinamic (DHTML);
- *RTFDoclet*, care generează documente în format RTF, care pot fi vizualizate cu MS Word;
- *DocLint*, care verifică corectitudinea comentariilor de documentație².

Totuși, dacă sunteți de părere că aceste doclet-uri nu vă sunt de folos, puteți să vă implementați propriul dumneavoastră doclet.

Implicit, `javadoc` utilizează doclet-ul standard și în majoritatea situațiilor vă veți mulțumi să îl utilizați pe acesta. Schimbarea acestuia cu unul la alegere se face prin specificarea opțiunii `-doclet` la apelul `javadoc`.

²Probabil că vă întrebați cum poate fi o documentație incorectă. După cum vom vedea în continuare, comentariile `javadoc` pot să conțină și anumite cuvinte cheie (etichete) care impun o anumită sintaxă. Corectitudinea comentariilor de documentație se referă la utilizarea corespunzătoare a acestor cuvinte cheie, și nicidecum la corectitudinea conținutului.

După cum precizam anterior, javadoc generează documentația API pe baza fișierelor sursă `.java`. Opțional, acestora li se mai pot adăuga și alte tipuri de fișiere:

- fișiere de comentare a pachetului. Fiecare pachet poate avea propriul lui comentariu de documentare, în care să fie precizate informații valabile pentru întreg pachetul. Pentru a realiza acest lucru, trebuie creat un fișier HTML cu numele `packages.html`, localizat în același director cu fișierele sursă `.java` care formează pachetul respectiv. Javadoc caută automat în fiecare director un fișier cu acest nume și îl cuprinde în documentație;
- fișiere de comentare per ansamblu. Există posibilitatea de a crea un comentariu la nivel de aplicație sau pentru mai multe pachete simultan. Crearea unui astfel de comentariu se face prin intermediul unui fișier denumit `overview.html`. Fișierul poate fi plasat oriunde, pentru că este unic în cadrul unei documentații generate, dar este de preferat să fie localizat în directorul rădăcină al fișierelor sursă `.java`. Ca și în cazul anterior, acest fișier trebuie scris în HTML;
- fișiere de alte tipuri (imagini, exemple de programe Java etc.), care sunt doar copiate de javadoc în directorul destinație. Aceste fișiere sunt necesare în cazul în care, de exemplu, comentariul de documentație din fișierul sursă face referire la o imagine etc.

Implicit, rezultatul generării documentației este o structură de directoare, deoarece javadoc folosește doclet-ul standard, care generează HTML. Directoarele respective conțin fișierele HTML generate propriu-zis de javadoc și eventualele fișiere (imagini, exemple de programe) care au fost doar copiate de javadoc. Pentru vizualizarea documentației generate, este de ajuns să aveți un *browser* de Internet (de exemplu, Internet Explorer sau Netscape Navigator) și să accesați pagina `index.html` din rădăcina directorului în care se află documentația generată. Accesarea acestei pagini oferă posibilitatea de a vedea rezultatul eforturilor de documentare a fișierelor Java. Documentația este prezentată sub forma a două *frame*-uri HTML (dacă nu a fost generată pentru pachete) sau sub forma a trei *frame*-uri HTML (în cazul pachetelor):

```
-----  
| C | Detaliu |  
|   |         |  
|   |         |  
-----
```

C = lista claselor pentru care s-a realizat documentația
 Detaliu = detaliile unei clase selectate în frame-ul din stânga

```
-----
| P | Detaliu |
|---|         |
| C |         |
-----
```

P = lista pachetelor pentru care s-a realizat documentația

C = lista claselor din pachetul selectat în frame-ul din stânga sus

Detaliu = detaliile unei clase selectate în frame-ul din stânga jos

Dacă pentru generarea documentației nu a fost folosit doclet-ul standard, atunci formatul documentației generate este altul, pentru vizualizarea ei fiind necesare, în mod firesc, alte aplicații.

Implicit, javadoc transformă comentariile în cod HTML, de aceea comentariile în sine pot conține cod HTML, pentru a permite o formatare mai bună. Prima propoziție a comentariului trebuie să reprezinte o descriere succintă dar completă a elementului comentat, deoarece această propoziție este folosită de javadoc pentru a crea un rezumat al descrierii clasei, interfeței, metodei sau pachetului respectiv. Se consideră sfârșitul propoziției simbolul punct (.), urmat de spațiu, *tab* sau apariția unui etichete speciale suportată de javadoc (detalii despre etichetele javadoc în secțiunea 11.3). Așadar, evitați comentariile de genul:

```
1 /**
2  * Modifica nota acordata de prof. unui elev.
3  */
```

Dacă doriți să folosiți totuși exprimarea anterioară, atunci apălați la următorul truc:

```
1 /**
2  * Modifica nota acordata de prof.&nbsp;unui elev.
3  */
```

B.11.3 Etichete javadoc

În cadrul comentariilor de documentație din interiorul fișierelor sursă, javadoc suportă anumite etichete, care sunt tratate special. Fiecare etichetă începe cu semnul @.

Lista completă a etichetelor javadoc este formată din următoarele elemente:

- `@author`
- `@deprecated`
- `@exception`
- `@param`
- `@return`
- `@see`
- `@serial`
- `@serialData`
- `@serialField`
- `@since`
- `@throws`
- `@version`
- `{@docRoot}`
- `{@link}`

Unele dintre acestea au fost deja utilizate în cadrul paragrafului B.3.3, la declararea claselor și a interfețelor, altele vor fi prezentate în continuare.

`@deprecated` adaugă în documentul generat un comentariu care arată că elementul respectiv (clasă, interfață, metodă etc) este învechit și că nu ar mai trebui folosit (chiar dacă ar putea să funcționeze). Totodată, trebuie precizat elementul care ar trebui folosit în locul celui demodat. Mod de utilizare: `@deprecated comentariu`.

`@see` adaugă informații ajutătoare, de genul "vezi metoda X". Informația apare la secțiunea *See also* a documentației generate, putând fi un simplu text sau un *link* către alte elemente. Poate fi utilizat în următoarele forme:

- `@see "comentariu"`, afișează text simplu
- `@see Numele link-ului`, afișează un *link* HTML

- `@see pachet.clasa#membru numeLink`, afișează un link HTML către documentația generată pentru membrul care aparține clasei din pachetul specificat.

`@since` afișează un text care indică versiunea aplicației software care a prezentat pentru prima dată elementul respectiv. De exemplu, `@since 1.3` denotă faptul că elementul a apărut în versiunea 1.3 a aplicației.

`@version` este utilizat în general cu argumentul `"%I%, %G%"`, care este convertit de javadoc în ceva similar cu `"1.39, 02/28/99"` (luna/zi/an), adică versiunea și data trecerii la versiunea respectivă.

`{@link pachet.clasa#membru numeLink}` se aseamănă cu eticheta `@see`, cu deosebirea că link-ul creat nu este afișat în secțiunea *See also*, ci chiar în cadrul textului în care este folosit.

B.11.4 Opțiunile comenzii javadoc

Opțiunile comenzii javadoc sunt destul de numeroase. Pentru a vedea lista completă a tuturor acestor opțiuni, executați

```
javadoc
```

în linia de comandă. Vom prezenta în continuare cele mai uzuale opțiuni:

```
-overview caleCatreFisier
```

Javadoc folosește această opțiune pentru a prelua comentariul de ansamblu al aplicației din fișierul specificat (`overview.html`). Calea către fișier este relativă în funcție de opțiunea `-sourcepath`.

```
-public
```

Generează documentație doar pentru clasele și membrii public.

```
-protected
```

Afișează doar clasele și membrii public și protected. Această opțiune este implicită.

```
-package
```

Afișează doar pachetele, clasele și membrii public și protected.

```
-private
```

Afișează toate clasele și toți membrii.

```
-doclet numeClasa
```

Este specificată clasa care execută doclet-ul folosit în generarea documentației. Calea către clasa de start a doclet-ului este specificată prin intermediul opțiunii `-docletpath`, urmată de calea respectivă.

```
-sourcepath listaCaiFisiereSursa
```

Este specificată calea către fișierele sursă (`.java`), atunci când numele de pachete sunt folosite ca parametri. De exemplu, dacă se dorește documenta-

rea unui pachet numit `com.mypackage`, ale cărui fișiere sursă sunt localizate la locația `c:\user\src\com\mypackage*.java`, atunci comanda ce trebuie executată arată astfel:

```
c:\>javadoc -sourcepath c:\user\src com.mypackage
-classpath listaCaiClase
```

Este specificată calea unde javadoc va căuta clasele referite (pentru detalii despre clasele referite, vezi secțiunile anterioare din această anexă).

`-Joptiune`

Lansează comanda `java` cu opțiunea specificată de argumentul `optiune`. De exemplu, pentru aflarea versiunii utilizate de javadoc, se execută comanda:

```
c:\> javadoc -J-version
```

Trebuie reținut faptul că nu există nici un spațiu între opțiunea `J` și argumentul `optiune`.

Opțiunile prezentate anterior pot fi utilizate de javadoc indiferent de doclet-ul folosit în generarea documentației. Implicit, javadoc utilizează doclet-ul standard, care generează documentație în format HTML. Acest doclet oferă un alt set de opțiuni, care vine în completarea celui anterior. Iată câteva dintre aceste opțiuni:

`-d`

Este specificat directorul destinație, unde javadoc va salva documentația HTML generată. Dacă această opțiune nu este prezentă, atunci documentația va fi creată în directorul curent.

`-version`

Include textul asociat cu eticheta `@version` în documentația HTML generată

`-author`

Include textul asociat cu eticheta `@author` în documentația HTML generată

`-splitindex`

Împarte alfabetic fișierul de index al documentației în mai multe fișiere, câte unul pentru fiecare literă.

`-windowtitle text`

Este specificat titlul care va fi plasat în tag-ul `<title>` din HTML (denumirea ferestrei *browser*-ului). Acest titlu devine titlul ferestrei de *browser*, vizibil în colțul din stânga sus al ferestrei.

`-doctitle text`

Este specificat titlul documentului.

`-header text`

Este specificat textul care va fi plasat la începutul fiecărui fișier de documentație generat.

-*footer text*

Este specificat textul care va apărea în dreapta meniului de navigare din josul paginii.

-*bottom text*

Este specificat textul care va apărea la finalul fiecărui fișier generat. În general, aceste note de subsol conțin informații legate de copyright.

-*nodeprecated*

Elementele demodate (pachete, clase, membrii etc.) sunt omise la generarea documentației.

B.11.5 Exemple simple de utilizare javadoc

Vom prezenta în final câteva exemple simple de folosire a utilitarului javadoc. Utilitarul poate fi rulat pe clase individuale sau pe pachete întregi de clase.

- Crearea documentației pentru unul sau mai multe pachete

Pentru a documenta un pachet, fișierele sursă din acel pachet trebuie să existe într-un director cu același nume ca și pachetul. Dacă pachetul este creat din mai multe nume, separate prin simbolul punct (.), fiecare nume reprezintă un director diferit. Astfel, toate clasele din pachetul `java.awt` trebuie să existe în directorul `java\awt\`.

Presupunem că directorul `c:\home\src\java\awt\` este directorul în care sunt create fișierele sursă iar directorul destinație pentru documentația generată este `c:\home\html`. Vom păstra această convenție pentru toate exemplele de acest gen din anexa curentă.

Documentația se poate crea în două moduri:

- din directorul corespunzător pachetului:

```
c:\> cd c:\home\src
c:\> javadoc -d c:\home\html java.awt
```

- din orice director:

```
c:\> javadoc -d c:\home\html -sourcepath
c:\home\src java.awt
```

- Crearea documentației pentru una sau mai multe clase

Analog cazului 1, avem două modalități disponibile:

- din directorul cu fișiere sursă:

```
c:\> cd c:\home\src\java\awt
c:> javadoc -d c:\home\html Canvas.java
Graphics*.java
```

Se generează documentația pentru clasa Canvas și pentru toate clasele al căror nume începe cu Graphics. Documentul HTML generat nu va mai avea trei frame-uri ca în cazul pachetelor, ci două, deoarece generarea s-a realizat pe clase și nu pe pachete, situație în care al treilea frame (cel cu listarea pachetelor) este inutil.

- din orice director:

```
c:\> javadoc -d c:\home\html
c:\home\src\java\awt\Canvas.java
c:\home\src\java\awt\Graphics*.java
```

- Crearea documentației pentru pachete și clase simultan

Combinarea celor două cazuri descrise anterior, conduce la:

```
c:\> javadoc -d c:\home\html -sourcepath
c:\home\src java.awt
c:\home\src\java\applet\Applet.java
```

- Crearea documentației prin specificarea în linie de comandă a unui fișier

Pentru a scurta sau simplifica o comandă javadoc, se pot specifica unul sau mai multe fișiere care conțin separat nume de fișiere sursă sau pachete, câte unul pe linie. La execuția comenzii, numele fișierelor trebuie precedat de simbolul @.

De exemplu, fie fișierul packages, cu următorul conținut:

```
com.package1
com.package2
com.package3
```

Se poate rula javadoc astfel (apidocs reprezintă numele directorului de destinație al documentației generate, iar home\src constituie directorul în care se află fișierul packages, dar și directorul com, cu subdirectoarele package1, package2, package3):


```
c:\> cd c:\home\src
c:> javadoc -d apidocs @packages
```

- Salvarea mesajelor de eroare generate de execuția javadoc într-un fișier separat (valabil doar pe Windows NT)

Pentru a redirecționa eventualele mesaje de eroare generate de execuția comenzii javadoc, se poate utiliza comanda:

```
c:\> javadoc -d docs java.lang >log.std 2>log.err
```

sau, forma restrânsă:

```
c:\> javadoc -d docs java.lang >javadoc.log 2>&1
```

Prima variantă redirecționează mesajele standard în fișierul log.std, iar pe cele de eroare în log.err, în timp ce în al doilea caz, ambele tipuri de mesaje sunt redirecționate către fișierul javadoc.log.

- Utilizarea din alte programe a comenzii javadoc

Se poate folosi o secvență de cod de genul:

```
1 void metoda ()
2 {
3     String [] javadocArguments = {
4         "-sourcepath ",
5         "c:\home\src ",
6         "-d",
7         "c:\home\html",
8         "java.awt"
9     };
10    com.sun.tools.javadoc.Main.main(javadocArguments);
11
12    //continuare cod...
13 }
```

Dezavantajul unei astfel de utilizări este că metoda main nu returnează nimic, de aceea nu se poate determina dacă javadoc s-a executat cu succes sau nu.

C. Definirea pachetelor Java. Arhive jar

Simplificați lucrurile atât cât este
posibil, dar nu mai mult.

Albert Einstein

C.1 Prezentare generală a pachetelor Java predefinite

Mediul de programare Java oferă o serie de clase predefinite pentru a veni în întâmpinarea programatorilor care doresc să realizeze aplicații la standarde profesionale. Scopul pentru care aceste clase au fost implementate de creatorii limbajului Java, a fost acela de a ușura pe cât posibil munca programatorilor, care nu mai sunt obligați în acest fel "să reinventeze roata". Programatorii folosesc aceste clase, dar își pot defini pe baza lor și propriile lor clase, specifice aplicațiilor pe care le realizează.

Mulțimea claselor oferite de limbajul Java poartă numele de Java 2 API (Application Programming Interface). API-ul Java oferă facilități foarte puternice, des utilizate de programatori. Putem spune cu certitudine că orice program Java utilizează din abundență API-ul Java. Dată fiind importanța lui, este evident că documentația API-ului capătă și ea o însemnătate deosebită. Documentația poartă numele Java 2 API Specification și poate fi găsită în cadrul documentației J2SDK.

Pentru o mai bună organizare, clasele oferite de limbajul Java au fost împărțite în pachete, având drept criteriu de departajare funcționalitatea oferită. Platforma Java este deosebit de solidă și oferă diverse facilități în diferite di-

recții, cum ar fi: *applet*-uri, aplicații de tip *desktop* folosind *AWT*, aplicații *Swing* bazate pe *JFC*, *RMI* (Remote Method Invocation), securizarea aplicațiilor, utilizarea bazelor de date, internaționalizarea aplicațiilor, crearea de arhive *JAR* și *ZIP*, suport pentru *CORBA*, etc. Multitudinea de direcții în care s-a dezvoltat Java este observabilă și în numărul mare de pachete pe care îl pune la dispoziția programatorilor: peste 70 de pachete. Numărul este impresionant și devine evident faptul că nu putem prezenta informații despre fiecare pachet în parte. Totuși, în cele ce urmează vom descrie cel mai des utilizate pachete, împreună cu câteva dintre cele mai reprezentative clase conținute.

Iată o listă a celor mai des utilizate pachete (în cadrul acestei lucrări sunt folosite doar clase din cadrul primelor cinci pachete):

- `java.lang`, care oferă clase fundamentale pentru limbajul Java;
- `java.util` conține clase pentru utilizarea colecțiilor, pentru manevrarea datelor calendaristice și a timpului, pentru internaționalizarea aplicațiilor etc.;
- `java.io`, care oferă modalități de citire/scriere a datelor prin intermediul fluxurilor de date, a serializării sau a fișierelor;
- `java.math`, care oferă clase specializate în calcul matematic;
- `java.text`, care oferă clase pentru manevrarea textului, a datelor calendaristice, a timpului și a mesajelor într-o manieră independentă de limba utilizată, fie ea engleză, franceză, română etc.;
- `java.net`, care pune la dispoziția programatorilor clase pentru implementarea aplicațiilor de rețea;
- `java.util.jar`, care oferă clase pentru citirea și scrierea fișierelor în format *JAR* (Java Archive), bazat pe formatul standard *ZIP*;
- `java.util.zip`, care oferă clase pentru citirea și scrierea de arhive în format *ZIP* sau *GZIP*.

Iată și câteva dintre cele mai reprezentative clase pentru fiecare dintre aceste pachete:

- `java.lang`
 - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, asociate tipurilor primitive respective: `byte`, `char`, `double` etc.;

- `Math`, asigură realizarea unor operații aritmetice elementare: modulul, funcții trigonometrice, exponențiale, maximul și minimul a două numere, aproximări etc.;
 - `String`, `StringBuffer`, pentru administrarea șirurilor de caractere;
 - `System`, oferă facilități de acces la fluxurile standard de intrare (`in`), ieșire (`out`) și eroare (`err`) etc.;
 - `Thread`, asigură facilități pentru lucrul cu fire de execuție în Java.
- `java.util`
 - `Calendar`, pentru administrarea datelor calendaristice;
 - `Date`, reprezintă o dată calendaristică în timp, cu precizie la nivel de milisecunde;
 - `Hashtable`, pentru reprezentarea tabelor de repartizare;
 - `LinkedList`, pentru administrarea listelor înlanțuite;
 - `Locale`, pentru reprezentarea unei zone politice, geografice sau culturale;
 - `ListResourceBundle`, `PropertyResourceBundle` sau `ResourceBundle`, pentru reprezentarea colecțiilor de resurse;
 - `Stack`, pentru administrarea unei stive (listă în format LIFO, LastIn-FirstOut);
 - `Vector`, implementează un șir de obiecte de dimensiune variabilă.
 - `java.io`
 - `BufferedReader`, citește text de la un flux de intrare a datelor, depunând informația citită într-un *buffer* pentru a realiza o citire eficientă a caracterelor, șirurilor, stringurilor;
 - `BufferedWriter`, scrie text către un flux de ieșire a datelor, depunând informația scrisă într-un *buffer* pentru a realiza o scriere eficientă a caracterelor, șirurilor, stringurilor;
 - `File`, care asigură gestionarea fișierelor, inclusiv a locațiilor unde acestea se găsesc;
 - `FileInputStream`, asigură citirea de date sub forma unor șiruri de *bytes* dintr-un fișier specificat;

- `FileOutputStream`, asigură scrierea de șiruri de *bytes* într-un fișier specificat;
- `java.math`
 - `BigDecimal`, `BigInteger`, care permit calculul cu numere foarte mari, care nu ar "încapa" în reprezentarea obișnuită a numerelor în Java.
- `java.text`
 - `ChoiceFormat`, permite atașarea de formate de mesaje în funcție de diverse valori ale unor variabile;
 - `DateFormat`, asigură formatarea datelor calendaristice și a timpului într-o manieră independentă de limba utilizată;
 - `MessageFormat`, permite formatarea mesajelor independent de limbă;
 - `NumberFormat`, permite formatarea numerelor, a procentelor și a sumelor de bani independent de limbă;
- `java.net`
 - `URLConnection`, oferă facilități pentru conectarea prin rețea la un server HTTP și adresarea unei cereri serverului respectiv;
 - `InetAddress`, pentru administrarea unei adrese IP (Internet Protocol);
 - `URL`, pentru administrarea unei adrese URL (Uniform Resource Locator), care indică o resursă Internet.
- `java.util.jar`
 - `JarFile`, pentru administrarea unui fișier JAR;
 - `JarInputStream`, oferă facilități de citire a conținutului unui fișier JAR;
 - `JarOutputStream`, oferă facilități de scriere a unui fișier JAR.
- `java.util.zip`
 - `ZipFile`, pentru administrarea unei arhive ZIP;

- `ZipInputStream`, oferă facilități de citire a conținutului unei arhive ZIP;
- `ZipOutputStream`, oferă facilități de scriere a unei arhive ZIP.

Dintre pachetele prezentate, două se detașează prin gradul foarte mare de utilizare, indiferent de natura aplicației care se crează. Este vorba despre pachetele `java.lang` și `java.util`. Așa cum am precizat, pachetul `java.lang` este importat implicit în orice aplicație Java, drept urmare programatorul poate folosi clasele sale fără a fi nevoie să le importe explicit.

C.2 Definirea de pachete de către utilizator

Secțiunea de față prezintă un exemplu simplu, dar detaliat, de creare și utilizare a propriilor pachete. Mediul de programare Java oferă spre utilizare programatorului, o mulțime de pachete împreună cu clasele aferente (de exemplu, `java.util`, `java.math` etc.). Dar pentru că această facilitate nu este întotdeauna suficientă, mediul Java permite definirea propriilor pachete. Cu alte cuvinte, programatorului Java îi este permis să își organizeze clasele definite de el în pachete proprii.

Exemplul ales utilizează trei pachete:

- `com.test.masini`
- `com.test.posesori`
- `com.test`

Primul pachet, `com.test.masini`, va îngloba clase cu funcționalitate apropiată, reprezentând diverse tipuri de mașini. Așadar, vom avea clasele `Dacia` și `Ferrari`. Al doilea pachet, `com.test.posesori`, va cuprinde clase care descriu posesorii automobilelor din primul pachet. Printre aceste clase se numără: `Persoana`, clasă asociată unei persoane fizice și `Firma`, clasă asociată unei persoane juridice (firme).

Programul principal, cu alte cuvinte clasa `ExempluPachete`, se va afla în pachetul `com.test` și va utiliza clase din celelalte două pachete. Pentru că definirea pachetelor are o influență decisivă asupra locației unde trebuie să fie salvate fișierele sursă ale acestei aplicații de test, considerăm directorul de lucru `c:\javawork` (pentru utilizatorii de Windows). Analog, utilizatorii altor sisteme de operare, vor putea implementa exemplul ales, considerând un director de lucru specific platformei respective (`/usr/local/javawork` este un exemplu pentru platforma Linux).

Câteva detalii despre pachete sunt necesare în acest moment al construirii aplicației. În primul rând, numele pachetelor trebuie să fie unice, pentru a evita eventualele conflicte care pot să apară la încărcarea claselor. Existența a două pachete cu același nume pune mașina virtuală Java în situația de a nu ști ce clase să încarce, din primul pachet sau din al doilea. Consecința imediată este faptul că nu se pot crea pachete cu același nume ca al pachetelor predefinite de mediul de programare Java (de exemplu `java.lang`, `java.util` etc). Sarcina de a găsi un nume unic pentru fiecare pachet poate fi ușurată dacă sunt respectate convențiile de programare Java (detalii în anexa B). Potrivit acestor convenții, putem crea pachete pornind de la domeniul Internet deținut, în cazul în care programatorul/firma deține un astfel de nume (de exemplu pachetul `ro.firmamea` provine de la domeniul `www.firmamea.ro`). Dacă nu există un domeniu Internet, este necesară crearea unei combinații de nume, care este puțin probabil să se repete. Afirmările anterioare sunt utile mai ales în cazul în care aplicațiile create de dumneavoastră sunt redistribuite către diverși utilizatori, care ar putea folosi la rândul lor alte pachete, de la alți furnizori. Exemplul nostru fiind unul de test, am ales ca nume de pachet `com.test`, nume care poate asigura unicitatea la nivelul calculatorului pe care realizăm aplicațiile Java de test, deși nu este personalizat cu informații cu caracter unic (de exemplu, numele persoanei sau al firmei care l-a creat).

Numele pachetului presupune crearea unei ierarhii de directoare conformă cu el. În cazul nostru, cele trei pachete existente `com.test`, `com.test.masini` și `com.test.posesori`, presupun crearea următoarelor directoare: `com`, `com\test`, `com\test\masini` și `com\test\posesori`, conform ierarhiei:

```
-com
  |
  - test
    |
    - masini
    |
    - posesori
```

Cele patru directoare sunt create în cadrul directorului de lucru stabilit anterior (`c:\javawork`). În fiecare dintre aceste directoare se află fișierele sursă ale claselor precizate anterior, în funcție de pachetul în care sunt definite:

- `Dacia.java`, `Ferrari.java` se găsesc în directorul `c:\javawork\com\test\masini`;

- `Persoana.java` și `Firma.java` în directorul `c:\javawork\com\test\posesori`;
- `ExempluPachete.java` în directorul `c:\javawork\com\test`.

Codul sursă al acestor fișiere este prezentat în continuare:

Listing C.1: Codul sursă al clasei `Dacia`

```
1 // Fișierul com\test\masini\Dacia.java
2 package com.test.masini;
3
4 public class Dacia
5 {
6     /** Modelul masinii (1310, 1300, Berlina, etc.). */
7     private String tip;
8
9
10    /** Creaza o Dacie cu modelul precizat. */
11    public Dacia(String tipul)
12    {
13        tip = tipul;
14    }
15
16    /** Returneaza tipul masinii. */
17    public String returneazaTip()
18    {
19        return tip;
20    }
21 }
```

Listing C.2: Codul sursă al clasei `Ferrari`

```
1 // Fișierul com\test\masini\Ferrari.java
2 package com.test.masini;
3
4 public class Ferrari
5 {
6     /** Modelul masinii (F50, etc.). */
7     private String tip;
8
9
10    /** Creaza un Ferrari cu modelul precizat. */
11    public Ferrari(String tipul)
12    {
13        tip = tipul;
14    }
15
16    /** Returneaza tipul masinii. */
17    public String returneazaTip()
```



```

18     {
19         return tip;
20     }
21 }

```

Listing C.3: Codul sursă al clasei Persoana

```

1 //Fisierul com\test\posesori\Persoana.java
2 package com.test.posesori;
3
4 import com.test.masini.*;
5 import java.util.*;
6
7 public class Persoana
8 {
9     /** Numele persoanei.*/
10    private String nume;
11
12    /** Adresa persoanei.*/
13    private String adresa;
14
15    /** Sir cu masinile Dacia detinute de persoana*/
16    private Vector masini;
17
18
19    /** Creaza o persoana cu numele si adresa specificate.*/
20    public Persoana(String numele, String adresa_)
21    {
22        nume = numele;
23        adresa = adresa_;
24        masini = new Vector();
25    }
26
27    /** Returneaza nume.*/
28    public String returneazaNume()
29    {
30        return nume;
31    }
32
33    /** Adauga o masina Dacia celor aflate deja in posesie.*/
34    public void adaugaDacia(Dacia masina)
35    {
36        masini.add(masina);
37    }
38
39    /** Returneaza masinile aflate in posesie.*/
40    public Vector returneazaMasini()
41    {
42        return masini;
43    }

```

44 }

Listing C.4: Codul sursă al clasei Firma

```
1 // Fisierul com\test\posesori\Firma.java
2 package com.test.posesori;
3
4 import java.util.*;
5 import com.test.masini.*;
6
7 public class Firma
8 {
9     /** Numele firmei.*/
10    private String nume;
11
12    /** Numar angajati.*/
13    private int numarAngajati;
14
15    /** Sir cu masinile aflate in posesie.*/
16    private Vector masini;
17
18
19    /**
20     * Creaza o firma specificand numele si
21     * numarul de angajati.
22     */
23    public Firma(String numele, int numarulAngajatilor)
24    {
25        nume = numele;
26        numarAngajati = numarulAngajatilor;
27        masini = new Vector();
28    }
29
30    /** Adauga un Ferrari celor aflate deja in posesie.*/
31    public void adaugaFerrari(Ferrari masina)
32    {
33        masini.add(masina);
34    }
35
36    /** Returneaza nume.*/
37    public String returneazaNume()
38    {
39        return nume;
40    }
41
42    /** Returneaza masinile aflate in posesie.*/
43    public Vector returneazaMasini()
44    {
45        return masini;
46    }
```

318

47 }

Listing C.5: Utilizarea claselor din pachetele definite

```

1 // Fisierul com\test\ExempluPachete.java
2 package com.test;
3
4 import com.test.masini.*;
5 import com.test.posesori.*;
6 import java.util.*;
7
8 public class ExempluPachete
9 {
10     public static void main(String[] args)
11     {
12         Persoana p = new Persoana("Mircea Ionescu", "");
13         Dacia d = new Dacia("1300");
14
15         //persoana P detine masina d
16         p.adaugaDacia(d);
17         Vector v = p.retorneazaMasini();
18
19         //afisarea tipurilor masinilor detinute de persoana
20         System.out.println("Tipurile de masini detinute de "
21             + p.retorneazaNume() + ": ");
22         for(int i = 0; i < v.size(); i++)
23         {
24             String tip = ((Dacia) v.get(i)).retorneazaTip();
25             System.out.print(tip + " ");
26         }
27         System.out.println("");
28
29         Firma firmaMea = new Firma("Sun", 13000);
30         Ferrari f = new Ferrari("F50");
31
32         //firma firmaMea detine masina f
33         firmaMea.adaugaFerrari(f);
34         v = firmaMea.retorneazaMasini();
35
36         System.out.println("Tipurile de masini detinute de "
37             + firmaMea.retorneazaNume() + ": ");
38         for(int i = 0; i < v.size(); i++)
39         {
40             String tip = ((Ferrari) v.get(i)).retorneazaTip();
41             System.out.print(tip + " ");
42         }
43     }
44 }

```

După ce clasele au fost create, următoarea acțiune constă în setarea vari-

abilei de sistem CLASSPATH, prezentată în cadrul capitolului 4.4.3. Variabila CLASSPATH trebuie astfel setată încât să conțină și directorul de lucru pentru aplicația noastră de test. Pentru aceasta, utilizatorii de Windows execută următoarea comandă în cadrul unui prompt MS-DOS:

```
set CLASSPATH=%CLASSPATH%;c:\javawork\
```

Pentru a verifica dacă variabila CLASSPATH a fost actualizată, se lansează în același prompt MS-DOS comanda:

```
echo %CLASSPATH%
```

Comanda afișează informații asemănătoare cu cele ce urmează:

```
.;c:\jdk\lib\tools.jar;c:\jdk\jre\lib\rt.jar;  
c:\javawork\
```

Dacă printre locațiile afișate se află directorul de lucru, atunci se poate trece la pasul următor.

Următorul pas constă în compilarea claselor definite în cadrul celor trei pachete care compun exemplul. Pentru ca procesul de compilare să fie mai ușor, se crează un fișier cu numele `fisiereSursa`, în directorul de lucru (`c:\javawork`), având următorul conținut:

```
com\test\ExempluPachet.java  
com\test\masini\Dacia.java  
com\test\masini\Ferrari.java  
com\test\posesori\Persoana.java  
com\test\posesori\Firma.java
```

Apoi, în același prompt MS-DOS în care s-a setat variabila CLASSPATH, se compilează fișierele sursă ale aplicației test, utilizând din directorul de lucru comanda:

```
javac @fisiereSursa
```

Este foarte important ca toate comenzile să fie executate în același prompt MS-DOS în care s-a setat variabila CLASSPATH, pentru că această setare este valabilă doar în cadrul acelui prompt. În momentul în care promptul respectiv este închis, se pierde modificarea realizată asupra variabilei CLASSPATH, revenindu-se la valoarea inițială. Pentru a face aceste modificări permanente, ele trebuie să fie efectuate în:

- fișierul `autoexec.bat` din directorul rădăcină, în cazul sistemelor Windows 95, 98, ME, urmate de repornirea calculatorului;
- secțiunea `User Variables` din `Control Panel` → `System` → `Environment`, în cazul sistemelor Windows NT, 2000.

Deoarece numărul de fișiere sursă este destul de mic pentru acest exemplu, folosirea comenzii anterioare este acceptabilă. Dar, dacă aplicația pe care o realizați are un număr mare de clase, atunci este recomandat să utilizați utilitarul `ant`, prezentat în anexa A, pentru a compila fișierele aflate într-o ierarhie de directoare.

Prin compilare, se crează fișierele `.class` respective, în cadrul aceluiași director cu cele sursă. Dacă se dorește separarea fișierelor sursă de cele `.class`, atunci comanda `javac` trebuie folosită cu opțiunea `-d`, urmată de directorul unde vor fi create fișierele `.class`. Trebuie menționat că acest nou director nu este automat precizat în variabila `CLASSPATH`. De aceea, dacă optați pentru această variantă, va trebui să adăugați personal directorul respectiv în cadrul variabilei `CLASSPATH`.

Exemplul construit în această secțiune se execută prin rularea în directorul de lucru a comenzii:

```
java com.test.ExempluPachet
```

Rezultatele afișate sunt:

```
Tipurile de masini Dacia detinute de Mircea Ionescu:
1300
Tipurile de masini Ferrari detinute de Sun:
F50
```

Pentru a înțelege în profunzime modul de lucru cu pachete, să vedem cum găsește Java pachetele și clasele definite de programator. Mai întâi, interpretorul Java (`java`) caută variabila de sistem `CLASSPATH`. Pornind de la rădăcină, interpretorul preia, pe rând, fiecare nume de pachet și înlocuiește fiecare simbol `"."` întâlnit cu simbolul `"\"` sau `"/` în funcție de sistemul de operare pe care rulează, pentru a genera o cale relativă unde vor fi căutate fișierele `.class`. În cazul nostru, pachetul `com.test.masini` este transformat în `com\test\masini`. Apoi, această cale relativă este concatenată la diferitele intrări din variabila de sistem `CLASSPATH`, de exemplu `c:\javawork` sau `c:\jdk\lib\tools.jar` etc. Fiecare dintre aceste concatenări nu reprezintă altceva decât căi absolute, unde interpretorul caută fișierele `.class` (adică

Dacia.class, Ferrari.class). Pentru exemplul nostru, interpretorul găsește fișierele anterioare în directorul `c:\javawork\com\test\masini`.

În acest moment se poate spune că primele pachete realizate de dumneavoastră sunt funcționale. Exemplul ales a fost însă unul simplu pentru a putea înțelege pașii necesari de la crearea pachetelor și până la rularea aplicației care le folosește. Cu siguranță că de-a lungul timpului nu vă veți opri aici și acest exemplu va fi urmat de altele, conținând ierarhii de pachete din ce în ce mai complexe.

C.3 Arhive jar (Java ARchives)

Arhivele jar permit gruparea mai multor fișiere într-un singur fișier arhivat. În mod obișnuit fișierele conținute de o arhivă jar sunt fișiere `.class`, împreună cu diverse resurse care pot fi asociate cu o aplicație (imagini, etc.). Nu există o interdicție în ceea ce privește tipul de fișiere conținute de o arhivă jar (acestea pot conține orice tip de fișiere, ca de altfel orice alt format de arhivare, cum ar fi `zip`, `rar`, `ace`, `arj` etc.), dar în general sunt folosite fișierele `.class`.

Arhivele jar oferă programatorilor anumite beneficii:

- *compresie*: arhivele jar permit compresarea fișierelor pentru a fi stocate mai eficient;
- *simplificarea distribuirii aplicațiilor*: aplicația creată este distribuită clienților sub forma unui singur fișier jar, care înglobează toate fișierele și resursele necesare funcționării aplicației;
- *micșorarea timpului de transfer*: pentru a fi descărcate fișierele conținute de un fișier jar, este nevoie doar de o singură conexiune HTTP și nu de câte o conexiune pentru fiecare fișier în parte;
- *portabilitate*: ca parte a platformei Java, arhivele jar sunt portabile, putând fi utilizate fără a fi modificate pe orice platformă cu suport Java;
- *extinderea funcționalității platformei Java*: mecanismul de extindere oferit de platforma Java permite extinderea funcționalității platformei prin adăugarea unor fișiere jar celor existente (printre arhivele existente în orice implementare Java se numără și `rt.jar`, ce conține API-ul Java predefinit, adică toată ierarhia de clase pe care mediul Java o oferă spre utilizare programatorilor). Prin adăugarea unor arhive noi, puteți să extindeți funcționalitatea platformei Java. Un exemplu este extensia *JavaMail*,

dezvoltată de Sun, care reprezintă un API utilizat în programele Java pentru transmiterea și recepționarea de email-uri;

- *versionarea pachetelor de clase*: un fișier jar poate cuprinde date despre fișierele pe care le conține, cum ar fi informații despre cel care a creat pachetele și despre versiunea pachetelor respective.

C.3.1 Utilizarea fișierelor .jar

Fișierele jar sunt arhivate pe baza formatului zip, deci au aceleași facilități ca orice alt tip de arhivă (compresie fără pierderea datelor etc.). Pe lângă aceste facilități de bază, fișierele jar au și alte funcționalități mai avansate (rularea unei aplicații în format jar este una dintre ele), a căror înțelegere este condiționată de familiarizarea cu operațiile fundamentale cu arhive jar: crearea unei arhive jar, vizualizarea, extragerea sau modificarea conținutului acesteia.

Operațiile de bază cu arhivele jar se realizează folosind o aplicație utilitară oferită de platforma Java. Această aplicație utilitară este în linie de comandă și este denumită jar, putând fi găsită în directorul bin al distribuției Java instalată pe calculatorul dumneavoastră. Pentru a testa prezența utilitarului pe calculatorul dumneavoastră, deschideți o fereastră de comandă (sau, altfel spus, un *prompt*) și rulați comanda jar. Implicit, vor fi afișate informații ajutoare despre această comandă, cum ar fi opțiunile de utilizare și câteva exemple.

Iată, pentru început, un rezumat al celor mai frecvente operații cu arhive jar:

- crearea unei arhive jar (cu alte cuvinte, "împachetarea" fișierelor și a directoarelor într-un singur fișier);
- vizualizarea conținutului unei arhive jar;
- extragerea conținutului unei arhive jar;
- extragerea doar a anumitor fișiere conținute într-o arhivă jar;
- modificarea conținutului unei arhive jar;
- rularea unei aplicații "împachetate" într-o arhivă jar;

C.3.2 Crearea unui fișier .jar

O arhivă jar este realizată utilizând aplicația utilitară jar în următoarea formă:

```
jar cf nume_fisier_jar nume_fisiere
```

Opțiunile și argumentele utilizate în această comandă au următoarea semnificație:

- opțiunea *c* indică faptul că este *creată* o arhivă *jar*;
- opțiunea *f* indică faptul că arhiva este creată într-un fișier. Este important de reținut faptul că opțiunile nu trebuie separate prin spațiu și că ordinea de apariție a lor nu contează;
- argumentul *nume_fisier_jar* reprezintă numele fișierului *jar* rezultat în urma arhivării. Poate fi folosit orice nume pentru a denumi un fișier *jar*. Prin convenție, fișierele au extensia *.jar*, deși acest lucru nu este obligatoriu;
- argumentul *nume_fisiere* reprezintă o listă de nume de fișiere și directoare, separate prin spațiu, care vor fi introduse în arhivă. Numele acestora poate să conțină și simbolul ***, pentru a face referire la mai multe fișiere în același timp. În cazul directoarelor, conținutul acestora este adăugat recursiv la arhivă;

Suplimentar, se pot folosi și alte opțiuni, pe lângă cele de bază (*cf*):

- opțiunea *v* afișează numele fiecărui fișier adăugat la arhivă, în timpul creării acesteia;
- opțiunea *0* indică faptul că arhiva nu va fi compresată;
- opțiunea *M* indică faptul că fișierul manifest asociat arhivei nu este creat (detalii despre fișierele manifest sunt oferite mai târziu în cadrul secțiunii curente);
- opțiunea *m* indică faptul că pot fi preluate dintr-un fișier manifest existent informații necesare arhivei care va fi creată. În acest caz, comanda are următoarea formă:

```
jar cfm nume_fisier_manifest nume_fisier_jar
      nume_fisiere
```

- opțiunea *-C* schimbă directorul curent în timpul execuției comenzii.

În continuare vom prezenta câteva exemple de creare a arhivelor jar, care utilizează fișierele .class ale aplicației de test cu mașini și posesori, prezentată în subcapitolul precedent. Aceste fișiere se află conform celor stabilite în subcapitolul precedent, în directorul `c:\javawork`, care are următoarea structură (pentru simplitate, sugerăm mutarea celorlalte tipuri de fișiere în alt director):

```
- com
  - test
    - masini
      - Dacia.class
      - Ferrari.class
    - posesori
      - Firma.class
      - Persoana.class
    - ExempluPachete.class
```

În primul exemplu de creare a arhivelor jar, în directorul `c:\javawork` se lansează dintr-o fereastră de comandă aplicația jar astfel:

```
jar cvf pachete.jar com
```

În fereastra de comandă se va vedea ceva asemănător cu textul următor:

```
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/test/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/Dacia.class(in = 364)
(out= 257) (deflated 29%)
adding: com/test/masini/Ferrari.class(in = 368)
(out= 259) (deflated 29%)
adding: com/test/posesori/(in = 0) (out= 0)(stored 0%)
adding: com/test/posesori/Persoana.class(in = 738)
(out= 432) (deflated 41%)
adding: com/test/posesori/Firma.class(in = 730)
(out= 439) (deflated 39%)
adding: com/test/ExempluPachete.class(in = 1602)
(out= 913) (deflated 43%)
```

Deoarece `com` este un director, utilitarul `jar` a adăugat recursiv toate directoarele și fișierele conținute în el. Rezultatul execuției acestei comenzi este

fișierul `pachete.jar`, aflat în directorul curent, în care a fost lansată comanda. Analizând textul afișat se poate observa că fișierul proaspăt creat este comprimat.

Există și posibilitatea de a crea o arhivă `pachete.jar` necomprimată. Pentru aceasta se lansează comanda:

```
jar cvf0 pachetel.jar com
```

Aceasta afișează următorul text:

```
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/test/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/Dacia.class(in = 364)
(out= 364) (stored 0%)
adding: com/test/masini/Ferrari.class(in = 368)
(out= 368) (stored 0%)
adding: com/test/posesori/(in = 0) (out= 0)(stored 0%)
adding: com/test/posesori/Persoana.class(in = 738)
(out= 738) (stored 0%)
adding: com/test/posesori/Firma.class(in = 730)
(out= 730) (stored 0%)
adding: com/test/ExempluPachete.class(in = 1602)
(out= 1602) (stored 0%)
```

Am ales să dăm o altă denumire arhivei din al doilea exemplu, pentru a putea compara dimensiunile celor două fișiere `jar` rezultate. Dacă cel comprimat, denumit `pachete.jar`, are 3.792 de bytes, cel necomprimat, `pachetel.jar`, are 5.180 de bytes.

Puteți să creați o arhivă `jar` folosind simbolul `*`, ca în exemplul următor:

```
jar cvf pachete.jar *
```

Cu ajutorul acestui simbol sunt adăugate în arhivă toate fișierele și (recursiv) directoarele din directorul curent.

Ultimul exemplu pe care îl prezentăm folosește opțiunea `-C`, de schimbare a directorului curent. Prin utilizarea acestei opțiuni, căile relative nu mai sunt păstrate în totalitate în cadrul fișierului `jar`. Pentru a observa acest comportament al utilitarului `jar`, lansați comanda:

```
jar cvf pachete.jar -C com\test\ .
```

În fereastra de comandă se va afișa:

```
added manifest
adding: masini/(in = 0) (out= 0)(stored 0%)
adding: masini/Dacia.class(in = 364) (out= 257)
(deflated 29%)
adding: masini/Ferrari.class(in = 368) (out= 259)
(deflated 29%)
adding: posesori/(in = 0) (out= 0)(stored 0%)
adding: posesori/Persoana.class(in = 738) (out= 432)
(deflated 41%)
adding: posesori/Firma.class(in = 730) (out= 439)
(deflated 39%)
adding: ExempluPachete.class(in = 1602) (out= 913)
(deflated 43%)
```

Cu alte cuvinte, în timpul execuției comenzii, directorul curent a fost schimbat în `com\test`, iar fișierele și directoarele au fost adăugate ca și cum comanda ar fi fost executată în acel director. Utilitatea acestei opțiuni apare în momentul în care se adaugă mai multe fișiere și directoare la o arhivă, deoarece atunci se poate schimba dinamic, dintr-o singură comandă, directorul curent de execuție a utilitarului `jar`.

C.3.3 Vizualizarea conținutului unui fișier `.jar`

Conținutul unui fișier `jar` poate fi vizualizat fără a-l extrage propriu-zis din interiorul fișierului. Comanda standard utilizată pentru a realiza acest lucru este:

```
jar tf nume_fisier.jar
```

Opțiunile și argumentele utilizate de această comandă au următoarea semnificație:

- opțiunea `t` indică faptul că se dorește vizualizarea conținutului unui fișier `jar`;
- opțiunea `f` indică faptul că în linia de comandă va fi specificat un nume de fișier, al cărui conținut va fi afișat;
- argumentul `nume_fisier.jar` reprezintă numele fișierului `jar` al cărui conținut va fi afișat.

Suplimentar, se poate utiliza și opțiunea `v` pentru a obține informații despre dimensiunea fișierelor care intră în componența arhivei `jar` și data în momentul ultimei modificări la care au fost supuse.

Ca exemplu, lansați în directorul `c:\javawork` următoarea comandă pentru a vedea care este conținutul arhivei `pachete.jar`, creată la secțiunea anterioară (primul exemplu din suită):

```
jar tf pachete.jar
```

Rezultatul execuției acestei comenzi este următorul:

```
META-INF/
META-INF/MANIFEST.MF
com/
com/test/
com/test/masini/
com/test/masini/Dacia.class
com/test/masini/Ferrari.class
com/test/posesori/
com/test/posesori/Persoana.class
com/test/posesori/Firma.class
com/test/ExempluPachete.class
```

După cum se poate observa, este afișat fiecare director și fișier conținut în fișierul `jar`. Directorul `META-INF` și fișierul `MANIFEST.MF`, conținut în acest director, reprezintă o componentă specială a oricărui fișier `jar` (paragraful C.3.5), fiind plasate în fișier la momentul creării acestuia. De asemenea, este util de reținut că toate directoarele și fișierele listate au căi relative, de tipul celei de mai jos: `com/test/masini/Dacia.class`.

C.3.4 Extragerea conținutului unui fișier `.jar`

Forma standard a comenzii cu ajutorul căreia se extrage conținutul unui fișier `jar` este următoarea:

```
jar xf nume_fisier.jar [nume_fisiere_arhivate]
```

Opțiunile și argumentele utilizate au semnificația:

- opțiunea `x` indică faptul că este *extras* conținutul unui fișier `jar`;
- opțiunea `f` specifică faptul că fișierul `jar` al cărui conținut va fi extras, este specificat în linie de comandă;

- argumentul `nume_fisier_jar` reprezintă numele fișierului al cărui conținut va fi extras;
- argumentul `nume_fisiere_arhivate` este un argument opțional (motiv pentru care apare între caracterele `[]`), reprezentând lista cu numele fișierelor care vor fi extrase din fișierul `jar`, separate prin spațiu. Dacă argumentul lipsește, atunci va fi extras întreg conținutul fișierului `jar`.

Când extrage conținutul unui fișier `jar`, utilitarul `jar` crează copii ale fișierelor și directoarelor care sunt extrase, reproducând structura de directoare pe care fișierele o au în cadrul arhivei. Toate aceste copii sunt create în directorul curent, fișierul `jar` rămânând intact. Însă, dacă există fișiere cu același nume în directorul curent, ele vor fi rescrise.

Ca exemplu, să extragem conținutul arhivei `pachete.jar`. Pentru aceasta, executați comanda următoare din directorul `c:\javawork`, într-o fereastră de comandă:

```
jar xf pachete.jar
```

Rezultatul execuției acestei comenzi sunt cele două directoare conținute în arhivă, împreună cu subdirectoarele și fișierele pe care le conțin. Aceste directoare sunt `com`, unde se află clasele aplicației de test, și `META-INF`, unde se găsește fișierul `MANIFEST.MF`. După cum se observă, toate fișierele și directoarele conținute au fost extrase, iar fișierul `pachete.jar` a rămas intact.

Pentru a extrage doar anumite fișiere din arhiva `jar`, trebuie să le specificați ca argument. Ca exemplu, executați comanda:

```
jar xf pachete.jar com/test/posesori/Firma.class
```

Această comandă crează ierarhia de directoare `com/test/posesori`, în cazul în care nu există în directorul curent, și apoi crează o copie a fișierului `Firma.class` în directorul `posesori`.

C.3.5 Fișierul manifest al unei arhive jar

Fișierul manifest este un fișier special al unei arhive `jar`, care conține informații despre fișierele arhivate. Implicit, fișierul are numele `MANIFEST.MF`.

Crearea unei arhive `jar` presupune, implicit, crearea unui fișier manifest, asociat cu respectiva arhivă. Fiecare arhivă `jar` are un singur fișier manifest, care este automat creat în directorul `META-INF` al arhivei. Numele fișierului manifest și al directorului în care se află acesta, nu pot fi modificate, deci întotdeauna fișierul manifest are locația `META-INF/MANIFEST.MF`.

Informațiile conținute într-un fișier manifest sunt de tipul "cheie: valoare". De exemplu, fișierul manifest al arhivei pachete.jar are următorul conținut:

```
Manifest-Version: 1.0
Created-By: 1.4.0 (Sun Microsystems Inc.)
```

Fișierul manifest conține două informații despre arhiva pachete.jar: versiunea fișierului manifest (specificată de cheia Manifest-Version) și numele platformei Java care a creat fișierul manifest respectiv (specificat de cheia Created-By).

Acesta este formatul standard al unui fișier manifest asociat unei arhive jar. Informațiile pe care le oferă diferă însă de la o arhivă la alta, în funcție de scopul arhivei. Cu alte cuvinte, pe lângă acestea mai pot apare și altele, care vor fi prezentate în continuare. Dacă rolul pe care îl are arhiva jar este doar cel standard al unei arhive, cum ar fi compresia datelor, nu trebuie să vă faceți probleme în legătură cu fișierul manifest, pentru că fișierul manifest nu are nici un rol în astfel de situații. Dacă, însă, doriți ca arhiva jar să prezinte funcționalități speciale (avansate), atunci va trebui să modificați conținutul fișierului manifest, dar pentru aceasta mai întâi va trebui să cunoașteți *ce* trebuie modificat în fișierul manifest și, după aceea, *cum* trebuie să-l modificați.

Pentru ca o aplicație "împachetată" într-un fișier jar, să poată fi executată folosind acest format, este necesară specificarea "punctului de intrare" al aplicației, adică numele clasei în care se află metoda `main()`, care este apelată în cazul în care aplicația nu este arhivată într-un fișier jar.

Specificarea numelui clasei respective se realizează utilizând cheia `Main-Class`. Pentru aplicația noastră de test, informația arată astfel:

```
Main-Class: com.test.ExempluPachete
```

Dacă aplicația dumneavoastră utilizează clase disponibile în alte arhive jar, atunci trebuie specificat acest lucru prin intermediul cheii `Class-Path`. Deși în cazul aplicației noastre nu se folosesc clase din alte arhive jar, iată un exemplu de prezentare a acestei informații:

```
Class-Path: test2.jar app/test3.jar
```

Pentru a reține informații referitoare la versiunea pachetelor conținute într-o arhivă jar, se pot utiliza chei speciale. Iată un exemplu, adaptat la datele existente în aplicația noastră:

```
Name: com/test/masini
```

```

Specification-Title: "Clase de masini"
Specification-Version: "1.0"
Specification-Vendor: "X Software"
Implementation-Title: "com.test.masini"
Implementation-Version: "versiune de test 0.12"
Implementation-Vendor: "X Software"

```

Un astfel de set de informații poate fi prezentat pentru fiecare pachet al aplicației.

C.3.6 Modificarea fișierului manifest al unei arhive jar

Modificarea fișierului manifest al unei arhive jar se realizează prin intermediul opțiunii `m` pe care o oferă utilitarul `jar`. Opțiunea `m` permite adăgarea în fișierul manifest a informațiilor dorite, în timpul operației de creare a arhivei jar.

Formatul standard al comenzii este:

```

jar cfm nume_fisier_manifest nume_fisier_jar
      nume_fisiere

```

Dat fiind faptul că prin această comandă se crează un fișier jar, opțiunile `cf` și argumentele `nume_fisier_jar` și `nume_fisiere` au aceeași semnificație cu cea descrisă la secțiunea de creare a unui fișier jar. Prin urmare, singurele lucruri care mai trebuie precizate sunt:

- opțiunea `m` specifică faptul că se vor adăuga anumite informații în fișierul manifest creat automat la crearea unei arhive jar;
- argumentul `nume_fisier_manifest` reprezintă numele fișierului din care vor fi preluate informațiile care vor fi adăugate în fișierul manifest.

Ca exemplu, vom recrea arhiva `pachete.jar`, adăugând fișierului manifest câteva informații cu caracter special. Astfel, în directorul curent, `c:\javawork`, se crează un fișier cu numele `info.txt`, având următorul conținut:

```

Main-Class: com.test.ExempluPachete

Name: com/test/masini
Specification-Title: "Clase de masini"
Specification-Version: "1.0"
Specification-Vendor: "CompaniaMea"

```

```
Implementation-Title: "com.test.masini"
Implementation-Version: "versiune de test 0.12"
Implementation-Vendor: "X Software"
```

După ce fișierul este salvat, se execută comanda:

```
jar cmf info.txt pachete.jar com
```

Rezultatul execuției acestei comenzi este arhiva pachete.jar. Pentru a vedea modificările efectuate în fișierul manifest, extrageți acest fișier din arhivă. Fișierul ar trebui să arate astfel:

```
Manifest-Version: 1.0
Main-Class: com.test.ExempluPachete
Created-By: 1.4.0 (Sun Microsystems Inc.)

Name: com/test/masini
Specification-Title: "Clase de masini"
Specification-Vendor: "X Software"
Implementation-Vendor: "X Software"
Specification-Version: "1.0"
Implementation-Version: "versiune de test 0.12"
Implementation-Title: "com.test.masini"
```

Se observă că noul fișier manifest este o combinație între fișierul standard creat de utilitarul jar și fișierul info.txt.

C.3.7 Modificarea conținutului unui fișier .jar

Conținutul unui fișier jar poate fi modificat folosind opțiunea u a utilitarului jar. Prin modificarea conținutului unui fișier jar se poate înțelege atât modificarea fișierului manifest asociat arhivei jar, cât și adăugarea unor noi fișiere în arhiva jar.

Forma standard a comenzii este:

```
jar uf nume_fisier.jar nume_fisiere
```

Opțiunile și argumentele utilizate au următoarea semnificație:

- opțiunea u semnifică faptul că arhiva jar va fi modificată;
- opțiunea f specifică faptul că numele fișierului care va fi modificat este precizat în linie de comandă;

- argumentul `nume_fisier_jar` reprezintă numele fișierului care este modificat (actualizat);
- argumentul `nume_fisiere` reprezintă o listă de fișiere ce vor fi adăugate în arhivă.

Dacă se utilizează opțiunea `m` (asemănător cu secțiunea anterioară, de modificare a fișierului manifest), atunci se poate modifica fișierul manifest al arhivei `jar`, concomitent cu adăugarea unor noi fișiere în arhiva `jar`.

Dacă, de exemplu, se dorește adăugarea în arhiva `pachete.jar` a unui nou fișier `new.gif`, aflat în directorul curent, atunci se execută comanda:

```
jar uf pachete.jar new.gif
```

De asemenea, se poate modifica fișierul manifest al arhivei `pachete.jar`, prin execuția următoarei comenzi (fișierul `info.txt` este cel definit la secțiunea anterioară):

```
jar umf info.txt pachete.jar
```

C.3.8 Rularea aplicațiilor Java "împachetate" într-o arhivă jar

Aplicațiile care sunt arhivate în cadrul unui fișier `jar` pot fi rulate folosind comanda:

```
java -jar nume_fisier_jar
```

Opțiunea `-jar` specifică faptul că aplicația ce va fi rulată este conținută într-o arhivă `jar`. Această comandă poate fi utilizată doar dacă în fișierul manifest al arhivei `jar` este specificat "punctul de intrare" al aplicației, prin intermediul cheii `Main-Class`. În consecință, pentru a putea rula o aplicație conținută într-o arhivă `jar`, trebuie mai întâi să modificați fișierul manifest pentru a introduce informația cerută.

În exemplul nostru, odată ce valoarea cheii `Main-Class` este specificată, se poate rula aplicația utilizând comanda:

```
java -jar pachete.jar
```

Rezultatul execuției acestei comenzi este următorul:

Tipurile de masini Dacia detinute de Mircea Ionescu:
1300
Tipurile de masini Ferrari detinute de Sun:
F50

În concluzie, secțiunea destinată arhivelor jar prezintă modalitatea de a realiza operații de bază cu arhive jar, precum și modalitatea de a rula aplicații conținute în cadrul unei astfel de arhive, înfățișând aceste caracteristici într-un mod simplu, la obiect, plin de exemple elocvente, pentru ca programatorul să poată surprinde facilitățile puternice pe care le oferă aceste arhive specifice platformei Java.

D. Internaționalizarea aplicațiilor. Colecții de resurse

Pentru un bărbat curajos, fiecare
țară reprezintă o patrie.

Proverb grecesc

Lumea în care trăim este mică și tinde să devină din ce în ce mai mică pe zi ce trece. În mare măsură, acesta este rezultatul implicării computerelor în viața noastră. Internetul a devenit în ultimii ani principalul mijloc de comunicare între oameni, pentru că oferă o modalitate foarte simplă și rapidă de comunicare. Emailul sau navigarea pe Internet nu mai reprezintă o necunoscută pentru majoritatea dintre noi. Practic, putem spune că trăim într-un imens sat, în care fiecare poate comunica simplu și rapid cu ceilalți. Efectul globalizării se simte și în industria IT, unde programatorii trebuie să considere ca piață de desfacere întreaga planetă. Ei sunt nevoiți să dezvolte aplicații care pot fi utilizate din Los Angeles până la Paris, fără ca acest lucru să implice un efort prea mare de programare. Cu alte cuvinte, sarcina lor este de a crea aplicații internaționalizate. Din acest punct de vedere, programatorii Java pot să fie liniștiți. J2SDK oferă o întreagă ierarhie de clase care permite adaptarea unei aplicații de la o limbă/cultură la alta, cu un efort minim de implementare. În Java, procesul este chiar mai simplu decât pare.

D.1 Timpul, numerele și datele calendaristice

Este foarte bine cunoscut faptul că, datorită diverselor popoare și culturi existente pe planeta noastră, nu există un standard acceptat la scară planetară pentru afișarea numerelor, a datelor calendaristice sau a timpului.

Să luăm ca exemplu următoarea dată: 5/3/02. Pentru cei ce locuiesc în Statele Unite, această dată reprezintă data de 3 mai 2002, în timp ce pentru cei ce locuiesc în Europa, aceasta reprezintă 5 martie 2002. După cum se poate observa, modul în care componentele unei date (zi, lună, an) sunt combinate și diferă de la o țară la alta. În Statele Unite, formatul este în stilul lună/zi/an, în timp ce în majoritatea țărilor din Europa formatul este zi/lună/an. Poate că nu este o diferență mare, dar nu trebuie subestimată importanța ei pentru utilizatorii aplicațiilor.

Formatele de date sunt poate cele mai simple. Ce se poate spune despre formatele de reprezentare a timpului sau cele de reprezentare a sumelor de bani (formate monetare) sau a numerelor? Răspunsul va fi descoperit pe parcursul acestei secțiuni a anexei.

Convențiile de formate variază de la o țară la alta, uneori semnificativ. Există situații în care aceste convenții variază chiar și în interiorul aceleiași țări. Iată câteva exemple:

- Timpul

Afișarea orelor, minutelor și secundelor pare să fie cea mai simplă dintre sarcini, dar nu este așa. Diferențe apar de exemplu la caracterul de separare a orelor de minute. Unele țări folosesc simbolul ":" (11:13 PM), în timp ce altele (Italia etc.) folosesc simbolul "." (23.13). De asemenea, cele 24 de ore ale zilei pot fi reprezentate ca jumătăți de zi de câte 12 ore (primul exemplu), sau cu ore cuprinse între 0 și 24 (al doilea exemplu);

- Datele calendaristice

Exemplele anterioare au arătat că poziția zilei, a lunii și a anului într-o reprezentare de dată calendaristică diferă de la o zonă la alta. Pe de altă parte, caracterul de separare diferă și el de la "/" în Statele Unite (de exemplu, 5/3/02), la "." în Germania (de exemplu, 03.05.02). De asemenea, numele zilelor și ale lunilor diferă de la o limbă la alta;

- Numerele

Primul lucru care atrage atenția în cazul numerelor este separatorul zecimal. Statele Unite folosesc simbolul "." (de exemplu, 3.55), dar alte țări (România etc.) preferă simbolul ",", (de exemplu 3,55). Pentru a îmbunătăți citirea numerelor mai mari, cifrele sunt grupate de obicei, obținându-se mii, milioane, miliarde etc. În Statele Unite acest simbol este "," (de exemplu, 3,403.45), iar în alte țări (România etc.) acest simbol este "." (de exemplu, 3.403,45). Sumele de bani sunt

reprezentate de asemenea diferit. Cei mai mulți sunt familiarizați cu semnul \$ folosit de Statele Unite și alte țări (de exemplu, \$3,400.00), dar există numeroase alte alternative.

De la lansarea ei, Java a fost catalogată drept soluția dezvoltării de aplicații internaționalizate. Dar până la apariția JDK 1.1, suportul pentru internaționalizare a fost redus. Începând însă cu versiunea anterior amintită, Sun a inclus în distribuția JDK pachetul `java.text` care conține clasele și interfețele pentru manevrarea textului în formate locale specifice diverselor culturi.

Înainte de a prezenta soluția Java pentru tratarea situațiilor anterioare, este cazul să prezentăm noțiunea de localizare (*engl.* locale) și clasa Java asociată, `Locale`.

O localizare poate desemna o regiune culturală, politică sau geografică. De exemplu, poate fi considerată regiunea din care provine un grup de persoane care vorbesc aceeași limbă. Din acest motiv, limba vorbită de aceste persoane este importantă pentru o localizare. Naționalitatea de asemenea, pentru că, de exemplu, deși americanii și britanicii vorbesc aceeași limbă, ei au totuși identități culturale diferite.

În Java, localizarea este reprezentată printr-o instanță a clasei `Locale` din pachetul `java.util`. O instanță a clasei `Locale` poate fi creată pentru orice limbă și orice țară. Localizarea se crează pe baza numelor limbii și a țării pentru care se realizează respectiva localizare. Ambele nume reprezintă de fapt abrevieri în conformitate cu standardele ISO. Codurile asociate limbilor reprezintă două litere mici ale alfabetului, ca de exemplu "en" pentru engleză, în timp ce codurile asociate țărilor reprezintă două litere majuscule, ca de exemplu "US" pentru Statele Unite.

Dacă doriți să vedeți lista completă a acestor abrevieri, vizitați următoarele site-uri:

- <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>, pentru codurile asociate limbilor;
- http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html, pentru codurile asociate țărilor.

Există însă anumite valori predefinite pentru unele țări (Canada, Franța, Germania, Italia, Statele Unite etc.) sau pentru unele limbi (engleză, franceză, italiană etc.), identificabile în cadrul clasei `Locale` prin intermediul atributelor statice ale clasei (`CANADA`, `FRANCE` pentru țări sau, respectiv, `ENGLISH`, `FRENCH`, pentru limbi).

În general, operațiile care necesită internaționalizarea (formatarea unei date calendaristice, de exemplu), folosesc ca argument o instanță a clasei `Locale`. Dacă acest lucru nu este specificat, atunci se folosește localizarea implicită.

Exemplul 1:

```
1 //foloseste localizarea implicita in
2 //operatii de internationalizare
3 Locale loc = Locale.getDefault();
4
5 //afiseaza limba asociata localizarii curente
6 System.out.println(loc.getDisplayLanguage());
7
8 //afiseaza tara asociata localizarii curente
9 System.out.println(loc.getDisplayCountry());
```

Dacă, de exemplu, localizarea implicită este reprezentată de limba engleză și Statele Unite, atunci secvența de cod anterioară afișează:

```
English
United States
```

Exemplul 2:

```
//creaza o localizare pentru Romania si limba romana
//primul parametru este codul ISO pentru limba romana
//al doilea parametru este codul ISO pentru numele tarii
Locale nl = new Locale("ro", "RO");
```

O localizare poate fi făcută implicită pentru o instanță a JVM-ului prin metoda `setDefault()`, ca în exemplul următor, în care `nl` este variabila din exemplul precedent):

```
//localizarea romana este setata ca fiind implicita
Locale.setDefault(nl);
```

sau

```
//localizarea germana este setata ca fiind implicita
Locale.setDefault(Locale.GERMAN);
```

Dacă o localizare nu este făcută implicită prin apelul metodei `setDefault`, atunci localizarea implicită este reprezentată de limba în care a fost instalată distribuția J2SDK.

Timpul poate fi formatat utilizând clasa `DateFormat`. De asemenea, cu ajutorul acestei clase se pot formata și datele calendaristice sau se pot construi timpul și datele calendaristice într-o formă independentă de limbă. Clasa oferă numeroase metode statice pentru obținerea timpului/datelor calendaristice, pe baza localizării implicite sau a uneia specificate explicit, și pe baza unui

număr de stiluri de formatare. Acestea includ opțiunile FULL, LONG, MEDIUM, SHORT. În general,

- opțiunea SHORT reprezintă data/timpul în forma: 10.12.02 sau 3:30 PM;
- opțiunea MEDIUM formatează data/timpul astfel: Dec 12, 2002;
- opțiunea LONG formatează data/timpul astfel: December 12, 2002 sau 3:30:55 PM;
- opțiunea FULL formatează data/timpul astfel: Thursday, December 12, 2002 AD sau 3:30:55 PM PST.

Pentru a putea formata timpul în localizarea curentă, este necesară obținerea unei instanțe, folosind metoda `getTimeInstance()` a clasei `DateFormat`. Formatarea propriu-zisă se realizează cu metoda `format()`, ca în exemplul:

```
1 //afisarea timpului forma scurta in localizarea implicita
2 DateFormat tf = DateFormat.getTimeInstance(DateFormat.SHORT);
3 System.out.println(tf.format(new Date()));
4
5 //afisarea timpului forma scurta in localizare italiana
6 tf = DateFormat.getTimeInstance(DateFormat.SHORT,
7                               Locale.ITALY);
8 System.out.println(tf.format(new Date()));
```

Rezultatul execuției acestei secvențe de cod arată astfel:

```
11:39 AM
11.39
```

Prima valoare afișată reprezintă timpul pentru localizarea implicită (engleză în cazul nostru), iar cea de a doua reprezintă timpul în localizare italiană. Diferența de formatare dintre cele două valori este cât se poate de clară.

Pentru a formata data calendaristică în localizarea curentă, se folosesc metodele `getDateInstance()` (pentru obținerea unei instanțe a clasei care permite administrarea datelor calendaristice) și `format()` (pentru formatarea propriu-zisă), ale clasei `DateFormat`:

```
1 //afisarea datei forma scurta in localizarea implicita
2 DateFormat tf = DateFormat.getDateInstance(DateFormat.SHORT);
3 System.out.println(tf.format(new Date()));
4
5 //afisarea datei forma scurta in localizare italiana
6 tf = DateFormat.getDateInstance(DateFormat.SHORT,
7                               Locale.ITALY);
8 System.out.println(tf.format(new Date()));
```

```

9
10 //afisarea datei forma scurta in localizare germana
11 tf = DateFormat.getDateInstance(DateFormat.SHORT,
12                                 Locale.GERMANY);
13 System.out.println(tf.format(new Date()));

```

Secvența afisează următoarele valori pentru 19 ianuarie 2002, corespunzătoare localizării implicite (engleză, în cazul nostru), localizării italiene (a doua variantă) și localizării germane (a treia variantă):

```

1/19/02
19/01/02
19.01.02

```

Analog poate fi folosită metoda `parse()`, care, spre deosebire de metoda `format()`, care transformă o dată calendaristică într-un string, realizează operația inversă, de creare a unei date calendaristice dintr-un string.

Numerele, procentele și sumele de bani se formatează utilizând clasa `NumberFormat`, într-un mod asemănător cu cele prezentate anterior.

Așadar, pentru a formata un număr într-o localizare la alegere se utilizează metodele `getInstance()` și `format()`:

```

1 //afisarea numerelor in localizarea implicita
2 NumberFormat nf = NumberFormat.getInstance();
3 System.out.println(nf.format(12500.5));
4
5 //afisarea numerelor in localizare germana
6 nf = NumberFormat.getInstance(Locale.GERMANY);
7 System.out.println(nf.format(12500.5));

```

Rezultatul este:

```

12,500.5
12.500,5

```

Prima valoare afișată este pentru localizarea implicită (engleză în cazul nostru), în timp ce cea de a doua este pentru localizarea germană.

Formatarea procentelor se realizează prin metodele `getPercentInstance()` și `format()`:

```

1 //afisarea procentelor in localizarea britanica
2 NumberFormat nf = NumberFormat.getPercentInstance(Locale.UK);
3 System.out.println(nf.format(0.3));

```

Rezultatul afișat în urma execuției acestei secvențe este:

```

30%
340

```


Formatarea monetară se realizează prin `getCurrencyInstance()` și `format()`:

```
1 //afisarea unei sume de bani in localizarea implicita
2 NumberFormat nf = NumberFormat.getCurrencyInstance();
3 System.out.println(nf.format(8599.99));
4
5 //afisarea unei sume de bani in localizare italiana
6 nf = NumberFormat.getCurrencyInstance(Locale.ITALY);
7 System.out.println(tf.format(8599.99));
```

Rezultatul poate considerat un pic surprinzător:

```
$8,599.99
L. 8.600
```

Surpriza o constituie faptul că pentru lira italiană nu se consideră subdiviziuni, sumele fiind rotunjite pentru a se obține valori întregi, acest lucru datorându-se faptului că moneda italiană este "slabă" în comparație cu celelalte monede. Ca și în cazul datelor calendaristice, se poate utiliza metoda `parse()` pentru a realiza operațiile inverse, de transformare a stringurilor în numere.

Probabil că sunt programatori care consideră în acest moment, că formatarea timpului, a datelor calendaristice și a numerelor sunt doar detalii de afișare a rezultatelor și nu au implicații foarte mari. Utilizatorii aplicațiilor s-ar putea să nu fie de acord cu această poziție, în special din cauza faptului că afișarea rezultatelor nu se face în conformitate cu așteptările lor. De aceea, programatorii trebuie să fie foarte atenți la acest detaliu pe care, poate, până acum nu l-au tratat cu îndeajuns de multă seriozitate, pentru a realiza o afișare a datelor într-o formă cât mai "prietenosă".

D.2 Colecțiile de resurse în internaționalizarea aplicațiilor

Când programatorii de la Sun au scris codul pentru cele două clase, `DateFormat` și `NumberFormat`, ei au fost cei care au decis ce localizări să suporte (engleză, franceză, germană etc.) și au oferit astfel modalitatea de a realiza conversiile de date pentru respectivele localizări. Deoarece variațiile datelor calendaristice și ale numerelor nu sunt foarte mari, ei au putut implementa clasele `DateFormat` și `NumberFormat`, în conformitate cu respectivele localizări, direct în biblioteca de clase (API) pe care Java o oferă.

Din păcate, lucrurile nu au mai fost la fel de simple în cazul mesajelor de tip text. Ar fi fost imposibil pentru programatorii de la Sun să anticipeze toate

mesajele de tip text folosite în aplicații (existente sau viitoare) și să ofere traduceri necesare. Din acest motiv, au lăsat această sarcină programatorilor Java care realizează aplicații internaționalizate. Aceasta înseamnă că programatorul Java este cel care decide ce localizări suportă aplicația, urmând ca tot el să ofere traduceri mesajelor text pentru fiecare localizare suportată de aplicația sa. Cu alte cuvinte, programatorul va crea o colecție de stringuri pentru o anumită localizare și le va utiliza în diverse locuri pe parcursul aplicației.

Pentru a realiza acest demers, implementatorii limbajului Java au pus la dispoziția programatorilor colecțiile de resurse (*engl.* resource bundles), cunoscute altfel și sub numele de pachete de resurse.

D.2.1 Un exemplu concret

Colecțiile de resurse înglobează toate informațiile specifice unei anumite localizări. După cum am văzut în paragraful 7.5 (pagina 212), fiecare resursă este asociată cu o cheie, care rămâne neschimbată pentru toate localizările. Valoarea asociată acestei chei se schimbă însă pentru fiecare localizare în parte, reprezentând de fapt traducerea în respectiva limbă a informației deținute de resursă.

Să considerăm următorul exemplu: dorim să realizăm o aplicație care afișează în diverse limbi, noțiunile de calculator, hard-disk, monitor și tastatură. Deoarece aceste denumiri se vor schimba în cadrul colecțiilor de resurse, fiind înlocuite cu traduceri lor, trebuie să le asociem cu niște chei care vor rămâne neschimbate. Presupunem că avem următoarele asocieri:

Cheie		Valoare
-----	-----	-----
calc		calculator
hdd		hard-disk
mon		monitor
tast		tastatura

Asocierile de mai sus formează o colecție de resurse, cu patru perechi de tipul cheie-valoare. Pentru fiecare limbă în care dorim să traducem aceste noțiuni, vom avea câte o colecție de resurse în care cheile vor fi aceleași cu cele anterioare, iar valorile vor reprezenta traduceri potrivite pentru fiecare noțiune. Vom considera că avem traduceri noțiunilor anterioare în trei limbi: engleză, germană și franceză. Prin urmare, vom avea alte trei colecții de resurse, câte una pentru fiecare limbă în care facem traducerea:

- pentru limba engleză

Cheie		Valoare
calc		computer
hdd		hard disk
mon		monitor
tast		keyboard

- pentru limba germană

Cheie		Valoare
calc		Computer
hdd		Platte
mon		Monitor
tast		Tastatur

- pentru limba franceză

Cheie		Valoare
calc		ordinateur
hdd		disque dur
mon		moniteur
tast		clavier

După cum se observă, faptul că aceste chei rămân neschimbate ne ajută să asociem corect fiecare noțiune cu traducerea ei. Astfel, noțiunea calculator are în primul tabel asociată cheia `calc`, care corespunde în următoarele trei tabele cuvintelor `computer`, `Computer`, `ordinateur`, cu alte cuvinte am obținut traducerea noțiunii în fiecare dintre cele trei limbi.

Un set de colecții de resurse formează un grup. Grupul are un nume, ales sugestiv de programator. Analog, fiecare colecție de resurse din cadrul grupului are un nume, construit în mod unic: numele grupului, urmat de numele limbii în care sunt traduse resursele, de numele țării și de *variant* (*variant* reprezintă o subregiune a unei țări). Ultimele două adăugări de nume sunt opționale. Numele limbii și cel al țării sunt reprezentate prin abrevieri de două litere, în conformitate cu standardele ISO. De exemplu, abrevierea pentru Canada este "CA", iar pentru limba franceză este "fr". În majoritatea cazurilor aceste nume sunt evidente.

Un alt aspect important legat de modul de construire a numelui unei colecții de resurse este că fiecare nume este separat de următorul prin simbolul "_".

Important de reținut este și faptul că există o colecție care are același nume cu grupul din care face parte. Aceasta este denumită *colecția implicită*.

Deși modalitatea de denumire a colecțiilor din cadrul unui grup poate părea dificilă, ea este de fapt destul de simplă. Pentru a verifica această afirmație, iată cum arată aceste denumiri în exemplul nostru: am denumit grupul de colecții Componente, ceea ce înseamnă că numele colecției implicite este tot Componente. Considerăm prima colecție prezentată (cea în limba română) ca fiind implicită. Cea de a doua colecție, cea cu traduceri în limba engleză se numește, conform regulii de denumire prezentată anterior, Componente_en. Analog, celelalte două colecții se numesc Componente_de și Componente_fr.

În continuare vom vedea cum implementează Java colecțiile de resurse sub forma unor subclase ale clasei abstracte ResourceBundle:

- ListResourceBundle
- PropertyResourceBundle

Vom analiza fiecare dintre cele două modalități mai detaliat în cele ce urmează.

D.2.2 Implementarea colecțiilor prin ListResourceBundle

ListResourceBundle conține resursele în cadrul unor clase Java standard, a căror denumire este dată de numele fiecărei colecții de resurse, așa cum a fost el stabilit prin aplicarea regulii de denumire. Aceste clase trebuie să suprascrie metoda `getContents()` și să ofere un șir care conține perechile de resurse (cheie, valoare) de tipul `String`. Pentru exemplul nostru, lucrurile stau astfel:

- în primul rând avem colecția implicită, implementată în modul următor (în fișierul `Componente.java`):

```
1 import java.util.*;
2
3 public class Componente extends ListResourceBundle
4 {
5     static final Object[][] contents = {
6         {"comp", "calculator"},
7         {"hdd", "hard-disk"},
8         {"mon", "monitor"},
9         {"tast", "tastatura"}
10    };
11
12    public Object[][] getContents()
```

```

13     {
14         return contents;
15     }
16 }

```

- apoi, avem implementarea colecției de resurse pentru limba engleză (în fișierul `Componente_en.java`):

```

1 import java.util.*;
2
3 public class Componente_en extends ListResourceBundle
4 {
5     static final Object[][] contents = {
6         {"comp", "computer"},
7         {"hdd", "hard disk"},
8         {"mon", "monitor"},
9         {"tast", "keyboard"}
10    };
11
12     public Object[][] getContents()
13     {
14         return contents;
15     }
16 }

```

- mai apoi, implementarea colecției de resurse pentru limba germană (în fișierul `Componente_de.java`):

```

1 import java.util.*;
2
3 public class Componente_de extends ListResourceBundle
4 {
5     static final Object[][] contents = {
6         {"comp", "Computer"},
7         {"hdd", "Platte"},
8         {"mon", "Monitor"},
9         {"tast", "Tastatur"}
10    };
11
12     public Object[][] getContents()
13     {
14         return contents;
15     }
16 }

```

- în final, implementarea colecției de resurse pentru limba franceză (în fișierul `Componente_fr.java`):

```

1 import java.util.*;
2
3 public class Componente_fr extends ListResourceBundle
4 {
5     static final Object[][] contents = {
6         {"comp", "ordinateur"},
7         {"hdd", "disque dur"},
8         {"mon", "moniteur"},
9         {"tast", "clavier"}
10    };
11
12    public Object[][] getContents ()
13    {
14        return contents;
15    }
16 }

```

Odată clasele implementate, ele pot fi utilizate cu succes, conform exemplului din paragraful D.2.4.

Până atunci, iată câteva detalii importante legate de implementarea colecțiilor de resurse prin această metodă. În primul rând, clasele au exact același nume cu cel stabilit la definirea colecțiilor: `Componente`, `Componente_en`, `Componente_de`, `Componente_fr`. În al doilea rând, informația de asociere a cheilor cu valorile corespunzătoare este conținută în interiorul claselor. Fiecare pereche cheie/valoare reprezintă un element al șirului `contents`. Din cauza faptului că datele sunt stocate în interiorul unor clase, apare un dezavantaj destul de important: clasele trebuie recompilate pentru orice modificare a valorilor. Este motivul pentru care această variantă este mai puțin utilizată de programatori.

O alternativă a acestei clase este clasa `PropertyResourceBundle`, care reține resursele în fișiere de proprietăți.

D.2.3 Implementarea colecțiilor prin `PropertyResourceBundle`

Clasa `PropertyResourceBundle` folosește un tip special de fișiere pentru a stoca informațiile de mapare a cheilor și a valorilor. Aceste fișiere sunt de fapt simple fișiere text, cu extensia `.properties`. Numele fiecărui fișier de acest tip este dat tot de regula de denumire a colecțiilor de resurse. Așadar, în exemplul nostru sunt patru fișiere de proprietăți cu următoarele denumiri:

- `Componente.properties`
- `Componente_en.properties`

- `Componente_de.properties`
- `Componente_fr.properties`

Formatul special al fișierelor de proprietăți apare datorită faptului că perechile cheie/valoare sunt stocate pe câte o linie separată. Fiecare linie este formată din cheie, urmată de semnul "=", iar în final este adăugată valoarea asociată cheii.

Pentru exemplul nostru, cele patru fișiere `.properties` au următorul conținut:

- fișierul `Componente.properties`:

```
comp=calculator  
hdd=hard-disk  
mon=monitor  
tast=tastatura
```

- fișierul `Componente_en.properties`:

```
comp=computer  
hdd=hard disk  
mon=monitor  
tast=keyboard
```

- fișierul `Componente_de.properties`:

```
comp=Computer  
hdd=Platte  
mon=Monitor  
tast=Tastatur
```

- fișierul `Componente_fr.properties`:

```
comp=ordinateur  
hdd=disque dur  
mon=moniteur  
tast=clavier
```

În acest moment, fișierele de proprietăți sunt gata pentru a fi utilizate.

D.2.4 Utilizarea colecțiilor de resurse

Indiferent că ați optat pentru prima metodă de implementare a colecțiilor de resurse (utilizând clasa `ListResourceBundle`), sau pentru cea de a doua (prin intermediul clasei `PropertyResourceBundle`), utilizarea colecțiilor de resurse se realizează în același mod. Așadar, dacă doriți să treceți de la o metodă la cealaltă, nu trebuie să modificați nimic în modul de utilizare a colecțiilor de resurse.

Această secțiune va arăta cât de elegant se face traducerea noțiunilor alese dintr-o limbă în alta. Pentru aceasta, vom considera următorul program de test (fișierul `UtilRB.java`):

```

1 import java.util.*;
2
3 public class UtilRB
4 {
5     public static void main(String[] args)
6     {
7         try
8         {
9             Locale.setDefault(new Locale("ro", "RO"));
10            ResourceBundle rb = ResourceBundle.getBundle(
11                "Componente", Locale.getDefault());
12
13            String s = rb.getString("comp");
14            System.out.println(s);
15
16            s = rb.getString("tast");
17            System.out.println(s);
18        }
19        catch (Exception e)
20        {
21            System.out.println("EXCEPTION: " +
22                e.getMessage());
23        }
24    }
25 }

```

Prezentat pe scurt, programul anterior setează localizarea implicită ca fiind cea română, iar apoi utilizează colecția de resurse atașată localizării implicite (cu alte cuvinte, colecția de resurse implicită, `Componente`), pentru a descoperi valorile atașate celor două chei `"comp"` și `"tast"`, valori pe care le afișează. Rezultatul execuției acestui program este:

```

calculator
tastatura

```

Analog se pot afișa valorile celorlalte chei.

Pentru a vedea adaptabilitatea foarte bună a colecțiilor de resurse la modificarea limbilor folosite, realizați în programul sursă anterior modificările necesare pentru ca programul să arate astfel:

```

1 import java.util.*;
2
3 public class UtilRB
4 {
5     public static void main(String[] args)
6     {
7         try
8         {
9             Locale.setDefault(new Locale("ro", "RO"));
10
11             ResourceBundle rb = ResourceBundle.getBundle(
12                 "Componente", Locale.ENGLISH);
13
14             String s = rb.getString("comp");
15             System.out.println(s);
16
17             s = rb.getString("tast");
18             System.out.println(s);
19         }
20         catch (Exception e)
21         {
22             System.out.println("EXCEPTION: " +
23                 e.getMessage());
24         }
25     }
26 }

```

Modificarea constă în faptul că s-a schimbat localizarea, adică limba curentă în care se fac afișările de date este cea engleză. Dat fiind faptul că localizarea curentă este cea engleză, la rularea programului, Java va prelua valorile cheilor "comp" și "tast" din colecția de resurse asociată limbii engleze, adică din colecția `Componente_en`. În urma execuției, vor fi afișate rezultatele:

```

computer
keyboard

```

Analog se poate obține traducerea noțiunilor în celelalte două limbi pe care le-am ales: germană și franceză. Pentru a realiza acest demers, trebuie doar să înlocuiți `Locale.ENGLISH` în programul anterior cu `Locale.GERMAN` și mai apoi cu `Locale.FRENCH`.

Un lucru important care trebuie amintit este legat de modul în care Java identifică pachetul de resurse pentru o anumită localizare. Colecția este căutată în cadrul grupului din care face parte, conform pașilor algoritmului următor,

căutarea încheindu-se în momentul în care este găsită colecția căutată sau s-a ajuns la colecția implicită.

1. `bundleName_localeLanguage_localeCountry_localeVariant`
2. `bundleName_localeLanguage_localeCountry`
3. `bundleName_localeLanguage`
4. `bundleName_defaultLanguage_defaultCountry_defaultVariant`
5. `bundleName_defaultLanguage_defaultCountry`
6. `bundleName_defaultLanguage`
7. `bundleName`

`bundleName` reprezintă numele grupului, iar `localeLanguage`, `localeCountry` și `localeVariant` se referă la limba, țara și regiunea localizării specificate, în timp ce `defaultLanguage`, `defaultCountry` și `defaultVariant` se referă la localizarea implicită.

Pentru a înțelege mai bine acest algoritm de căutare a colecțiilor de resurse, vom arăta cum se realizează căutarea în exemplul nostru. Mai întâi să identificăm numele din algoritm: `bundleName` este în exemplul nostru `Componente`, `localeLanguage` este fie `en` (engleza), fie `de` (germana), fie `fr` (franceza), în timp ce `localeCountry` și `localeVariant` lipsesc. Pe de altă parte, `defaultLanguage` este `ro` (româna), în timp ce `defaultCountry` și `defaultVariant` lipsesc de asemenea.

Dacă, de exemplu, dorim să căutăm colecția de resurse pentru limba germană, trebuie să căutăm colecția cu numele `Componente_de`, cu alte cuvinte `bundleName` ia valoarea `Componente`, iar `localeLanguage` ia valoarea `de`. Pentru că nu avem în această situație `localeCountry` sau `localeVariant`, algoritmul va fi executat până la pasul 3, când se oprește pentru că a găsit colecția cu numele `Componente_de`.

Într-un mod asemănător, dacă dorim să căutăm colecția de resurse pentru limba franceză vorbită în Canada (`new Locale("fr", "CA")`), Java va căuta colecția cu numele `Componente_fr_CA`. Executând pașii algoritmului anterior, ne vom opri la pasul 3, pentru că nu există o colecție numită `Componente_fr_CA` (și din acest motiv nu ne putem opri la pasul 2), dar există una numită `Componente_fr`. Așadar, în această situație vom avea traduceri din limba franceză standard, și nu traduceri pentru dialectul de limbă franceză vorbit în Canada.

Forțând un pic nota, dacă dorim să căutăm colecția de resurse pentru limba italiană (`Locale.ITALIAN`), Java caută colecția de resurse cu numele `Componente_it`, dar cum nu avem nici o componentă cu nume asemănător, căutarea se va încheia la pasul 7, la colecția implicită `Componente`, traducerea fiind preluată din această colecție. Prin urmare, dacă nu avem o colecție asociată unei localizări specificate, traducerea vor fi oferite din colecția implicită.

Folosind tehnicile prezentate în primele două secțiuni ale acestei anexe, se pot construi aplicații simple care afișează utilizatorului date în orice localizare. Datele pot fi valori ale timpului, date calendaristice, numere sau simple mesaje statice, cum ar fi cele prezentate anterior. Dacă dorim să folosim mesaje dinamice (de genul: "x fișiere au fost șterse"), lucrurile se modifică destul de mult. Secțiunea următoare arată cum trebuie tratate mesajele de acest tip.

D.3 Internaționalizarea mesajelor dinamice

Din perspectiva programatorului care rescrie aplicații într-o altă limbă, cuvintele sunt cea mai simplă parte ce trebuie implementată. De exemplu, când se traduc cuvintele din engleză în germană, "Yes" devine "Ja", "No" devine "Nein" și așa mai departe. Chiar și frazele simple, de tipul "Do you want to save the modifications?", pot fi înlocuite în întregime cu traducerea potrivite. Dacă traducerea s-ar rezuma doar la exemple simple, de tipul celor menționate anterior, atunci nu ar fi probleme. Dar ce facem dacă suntem nevoiți să traducem un mesaj de tipul: "You copied 2 files"? Spre deosebire de mesajele anterioare, acesta este un mesaj dinamic, conține informații care se modifică în funcție de alte operații.

O metodă de rezolvare ar putea fi împărțirea mesajului în mai multe părți, în așa fel încât partea dinamică a mesajului să fie despărțită de cea statică. Cu alte cuvinte să împărțim mesajul în trei părți: "You copied", "2" și "files". În acest fel, partea dinamică a mesajului ("2") este separată de cea statică. Apoi, ar urma ca cele două părți statice să fie preluate din colecții de resurse, unde ar fi traduse separat. Practic, codul ar fi asemănător cu:

```
1 ResourceBundle rb = ResourceBundle.getBundle("Strings");
2 String str = rb.getString("first") + nFiles +
3           rb.getString("second");
```

unde "first" și "second" ar fi cheile asociate celor două părți statice ale mesajului, iar `nFiles` ar fi numărul de fișiere ce va fi copiat.

Din păcate, abordarea precedentă nu este universal valabilă, deoarece este posibil ca în alte limbi ordinea cuvintelor să nu mai fie aceeași. Deși noi am considerat că elementele care compun textul au o anumită ordine, există limbi în care această ordine nu poate fi respectată. De exemplu, în limba germană verbele se pun la finalul propoziției.

Din fericire, există o soluție pentru această problemă. Trebuie să creăm un șablon corect din punct de vedere sintactic pentru fiecare localizare, iar apoi la executarea aplicației șablonul este combinat cu partea dinamică a mesajului (altfel spus, cu parametrii). Abilitatea de a combina șablonul cu parametrii este oferit în Java prin intermediul clasei `MessageFormat`.

Primul pas care trebuie realizat este crearea câte unui șablon pentru fiecare localizare suportată de aplicație. Șablonul presupune înlocuirea părților dinamice ale mesajului, cu parametrii, ca în exemplul:

```
"You copied 2 files" --> "You copied {0} files"
```

În cazul nostru, singura parte dinamică a mesajului este "2" și a fost înlocuită cu parametrul "{0}". Dacă ar fi existat mai multe părți dinamice în cadrul mesajului, parametrii ar fi fost {1}, {2} etc.

Considerăm șablonul anterior ca fiind asociat localizării implicite (în cazul nostru, localizarea engleză). De asemenea, considerăm și șablonul pentru localizarea română:

```
"Ati copiat {0} fisiere"
```

La executarea aplicației, parametrul {0} este înlocuit în cadrul șablonului, cu numărul de fișiere copiate, așa cum a fost el calculat în cadrul aplicației.

Cele două șabloane, câte unul pentru fiecare dintre cele două localizări, vor fi preluate din colecții de resurse și folosite pentru a crea mesajele dinamice. Considerăm că numele grupului de colecții este "Messages", iar cheia asociată șablonului este "msg".

Colecția de resurse implicită este următoarea (`Messages.properties`):

```
msg=You copied {0} files
```

Pe de altă parte, colecția de resurse pentru limba română arată astfel (fișierul `Messages_ro.properties`):

```
msg=Ati copiat {0} fisiere
```

Pentru localizarea implicită, secvența de cod arată astfel:

```

1 ResourceBundle rb = ResourceBundle.getBundle("Messages");
2
3 MessageFormat mf = new MessageFormat(rb.getString("msg"));
4 Object[] args = {new Integer(2)};
5
6 System.out.println(mf.format(args));

```

La execuția codului va fi preluat din colecția implicită șablonul "You copied {0} files". Prin apelului metodei `format()`, șablonul este combinat cu valorile aflate în variabila `args`. Practic, parametrii șablonului sunt înlocuiți cu valorile de pe pozițiile corespunzătoare din șirul `args`. În cazul nostru, parametrul {0} este înlocuit cu `args[0]`, adică cu elementul 2, obținându-se mesajul formatat "You copied 2 files".

Pentru obținerea traducerii acestui mesaj în limba română, modificați prima linie a secvenței de cod anterioare, astfel încât să fie de forma:

```

ResourceBundle rb = ResourceBundle.getBundle("Messages",
                                             new Locale("ro", "RO"));

```

Analog celor prezentate anterior, se va obține un mesaj de forma "Ați copiat 2 fișiere".

Exemplele anterioare folosesc cele mai simple forme de parametri în cadrul unui șablon. Java oferă posibilitatea de a realiza formatare mult mai complexe.

Un parametru este format de fapt din trei câmpuri separate prin simbolul ", ":

- Primul câmp reprezintă un număr, care indică poziția din cadrul șirului de obiecte, a obiectului cu care va fi înlocuit parametrul la formatare. De exemplu, parametrul 0 indică faptul că va fi înlocuit cu elementul de pe poziția 0 din șir, parametrul 1 cu elementul de pe poziția 1 etc.;
- Al doilea câmp este opțional și se referă la tipul obiectelor formate. Poate avea una din următoarele valori: `time`, `date`, `number`, `choice`;
- Al treilea câmp este de asemenea opțional și se referă la stilul de formatare a obiectelor. Dacă formatul stabilit la punctul anterior este `time` sau `date`, atunci stilul poate avea valorile: `short`, `medium`, `long`, `full` sau un stil de dată definit de utilizator. Dacă formatul este `number`, atunci stilul poate avea valorile: `currency`, `percentage`, `integer`, sau un stil de număr definit de utilizator.

Iată câteva exemple de parametri:

```
{0}           - formateaza elementul de pe pozitia
```

	0 din sir
{0,time}	- formateaza elementul de pe pozitia 0 din sir ca reprezentand timpul
{0,date,short}	- formateaza elementul de pe pozitia 0 din sir ca fiind o data calendaristica in format scurt
(0,number,percent)	- formateaza elementul de pe poz. 0 din sir ca fiind un procent

Formatul `choice` este mai special și îl vom descrie în continuare. Opțiunea `choice` (din lista de opțiuni pentru câmpul al doilea al unui parametru) permite atașarea de mesaje unui anumit domeniu de valori. Opțiunea este utilă pentru că poate modifica mesajele în funcție de anumite valori ale parametrilor. Dacă în exemplul nostru, valoarea parametrului ar fi fost 1 sau 0, mesajul ar fi arătat astfel:

"You copied 1 files"

sau

"You copied 0 files"

Prima propoziție nu este corectă din punct de vedere gramatical (`files` este la plural) și Java ne ajută să evităm o astfel de situație, prin folosirea opțiunii `choice`. Opțiunea ne oferă posibilitatea de a modifica mesajul în funcție de valoarea parametrului, astfel încât să avem următoarele mesaje:

- dacă parametrul are valoarea 0, atunci mesajul este "You copied no files.";
- dacă parametrul are valoarea 1, atunci mesajul este "You copied 1 file.";
- dacă parametrul are valoarea > 1 , atunci mesajul este "You copied N files.", unde N este un număr.

Sintaxa opțiunii `choice` este descrisă în continuare. Opțiunea este specificată ca un set de alternative, separate prin simbolul `"|"`. O alternativă constă dintr-o limitare numerică și un mesaj. Alternativele sunt ordonate după limitarea numerică. Limitarea reprezintă un număr `double`, care poate fi specificat în două moduri:

- $N\#$, ce înseamnă că pentru toate valorile mai mari decât N , inclusiv, mesajul este valabil;
- $N<$, ce înseamnă că pentru toate valorile mai mari decât N , exclusiv, mesajul este valabil.

Folosind opțiunea `choice`, șablonul mesajului nostru arată astfel:

```
"You copied {0,choice,0#no files|1#1 file|
1<{0} files}"
```

Deși pare destul de complicată, la o privire mai atentă se poate observa că utilizarea `choice` nu este chiar atât de dificilă. Șablonul ne dezvăluie că formatarea se aplică parametrului 0, adică elementului de pe poziția 0 din șirul de elemente, că este o formatare de tipul alegere (*engl.* `choice`) cu trei alternative, separate prin două simboluri `"|"`. Prima alternativă indică faptul că mesajul `"no files"` va fi afișat pentru valori ale parametrului ≥ 0 și < 1 (conform limitării numerice a celei de a doua alternative), adică pentru valoarea 0. Asemănător, alternativa a doua indică faptul că mesajul `"1 file"` este afișat pentru o valoare a parametrului între ≥ 1 (limitarea numerică a alternativei a doua) și ≤ 1 (limitarea numerică a alternativei a treia), adică pentru valoarea 1 a parametrului. Alternativa a treia indică faptul că pentru valori ale parametrului > 1 , se afișează mesajul `"N files"`, unde N este valoarea parametrului.

După cum se poate observa, limitările numerice ale alternativelor sunt ordonate crescător (0, 1, 1).

Cu prezentarea formătărilor mesajelor dinamice se încheie și ultima etapă care trebuie parcursă pentru a putea crea aplicații internaționalizate la standarde profesionale. Ajuns la acest nivel, programatorul Java trebuie să conștientizeze faptul că internaționalizarea aplicațiilor nu mai reprezintă o irosire a timpului, ci chiar o necesitate.

E. Resurse Java

Important este să nu încetezi
niciodată să îți pui întrebări.

Albert Einstein

E.1 *Site-ul Sun Microsystems*

Pagina principală a tuturor resurselor Java este `http://java.sun.com`, realizată de creatorii limbajului Java, unde veți găsi informații complete despre acest limbaj. *Site-ul* reprezintă o colecție impresionantă de documentații, tutoriale, cărți despre Java. Sugestia noastră este ca în situația în care aveți nevoie de informații suplimentare despre limbajul Java, să începeți căutarea dumneavoastră aici.

Iată câteva adrese utile în cadrul *site-ului*:

- `http://java.sun.com/docs` - documentații Java;
- `http://java.sun.com/products/jdk/faq.html` - răspunsuri la întrebări adresate frecvent;
- `http://java.sun.com/docs/books` - cărți despre limbajul Java;
- `http://java.sun.com/docs/books/tutorial` - tutorialul complet al limbajului Java;
- `http://java.sun.com/j2se` - pagina principală a platformei J2SE;
- `http://java.sun.com/products` - pagina de informații despre principalele produse ale Sun Microsystems.

E.2 Tutoriale despre limbajul Java disponibile pe Internet

Tutoriale Java pot fi găsite la următoarele adrese:

- <http://www.artima.com>
- <http://www.zdnet.com>
- http://directory.google.com/Top/Computers/Programming/Languages/Java/FAQs,_Help,_and_Tutorials/Tutorials
- <http://developer.java.sun.com/developer/onlineTraining/Programming>
- <http://www.javacoffeebreak.com/tutorials>
- <http://webreference.com/programming/java/tutorials.html>
- <http://freewarejava.com>
- <http://www.1netcentral.com/online-tutorials.html>
- <http://www.programingtutorials.com/tutorials.asp?id=Java>
- <http://javaboutique.internet.com>

E.3 Cărți despre limbajul Java disponibile pe Internet

Marea majoritate a *site*-urilor care prezintă cărți despre limbajul Java oferă doar o mică descriere a acestora, pentru a crea o impresie de ansamblu celor care doresc să cumpere cărțile respective. Există însă și cărți complete disponibile gratuit. Pentru ambele categorii vom prezenta câteva adrese utile:

- <http://java.sun.com/docs/books>
- <http://www.informit.com>

- <http://gshulkin.hypermart.net/books/FreeJavaBook.htm>
- <http://freewarejava.com/books/index.shtml>
- <http://www.better-homepage.com/java/java-books.html>
- <http://www.mindview.net>, unde poate fi găsită gratuit una dintre cele mai bune cărți despre Java, "Thinking in Java", scrisă de Bruce Eckel.

E.4 Reviste online de specialitate

Apariția Internetului a însemnat și o prezentare a revistelor într-un nou format, cel electronic, mult mai accesibil decât cel standard (cel puțin din punct de vedere al distanței, deoarece oricine poate citi o revistă care apare în SUA chiar din momentul apariției ei). Comunitatea IT se bucură de prezența a numeroase reviste online de prestigiu, care oferă atât informații generale din domeniul IT cât și despre limbajul Java în special:

- <http://www.javaworld.com> - liderul revistelor online care tratează limbajul Java
- <http://www.pcmag.com>
- <http://www.slashdot.org> - revistă online care prezintă informații generale din domeniul IT și nu numai
- <http://www.eneews.com>
- <http://www.infoworld.com>

E.5 Liste de discuții despre limbajul Java

Pentru a comunica online (prin email) cu alți programatori, pe tema limbajului Java, vă recomandăm înscrierea la lista de discuții *Advanced Java*, găzduită de Universitatea Berkeley:

<http://lists.xcf.berkeley.edu/mailman/listinfo/advanced-java>

E.6 Alte resurse Java

În final, vă prezentăm o listă de *site-uri* unde puteți găsi diverse alte informații despre limbajul Java:

- <http://www.ibiblio.org/javafaq>
- <http://www.gamelan.com>
- <http://www.apl.jhu.edu/~hall/java>
- <http://www.web-hosting.com/javalinks.html>

Bibliografie

- [Andonie] R. Andonie, I. Gârbacea - Algoritmi și Calculabilitate, Computer Press Agora, 1995
- [Balanescu] T.Bălănescu - Metoda Backtracking, Gazeta de informatică, nr 2/1993, pag. 9-18
- [Boian] F.M. Boian - De la aritmetică la calculatoare, Presa Universitară Clujeană, 1996, ISBN 973-97535-5-8
- [Cormen] T.H. Cormen, C.E. Leiserson, R.R. Rivest - Introducere în Algoritmi, Computer Press Agora, 2000, ISBN 973-97534-3-4
- [Danciu] D.Danciu, S.Dumitrescu - Algoritmica și Programare, Curs și Probleme de Seminar, Reprografia Univ. Transilvania, 2002
- [Eckel] B. Eckel - Thinking in Java, <http://www.mindview.net/Books/TIJ/>
- [Horowitz] E. Horowitz, S. Sahni - Fundamentals of Computer Algorithms, Computer Science Press, ISBN 3-540-12035-1
- [Norton] P. Norton, W. Stanek - Peter Norton's Guide to Java Programming, Sams.net Publishing, 1996, ISBN 1-57521-088-6
- [Roman] Ed Roman - Mastering Enterprise JavaBeans, Wiley Computer Publishing, 1999, ISBN 0-471-33229-1
- [Sorin] T.Sorin - Tehnici de programare, manual pentru clasa a X-a, L&S Infomat, 1996
- [Tomescu] I. Tomescu - Data Structures, Bucharest University Press, Bucharest, 1997

- [Vanderburg] G. Vanderburg - Tricks of the Java Programming Gurus, Sams.net Publishing, 1996, ISBN 1-57521-102-5
- [Weiss] M.A. Weiss - Data Structures and Problem Solving Using Java, Addison-Wesley, 1998, ISBN 0-201-54991-3
- [Sundsted] T. Sundsted - Internationalize dynamic messages, JavaWorld article (www.javaworld.com)
- [JavaTutorial] Sun Microsystems - The Java Tutorial (java.sun.com/docs/books/tutorial)
- [Pal] G. Pal - Exceptions in Java. Nothing exceptional about them, JavaWorld article (www.javaworld.com)
- [JavaDocs] Sun Microsystems - Java docs (java.sun.com/docs)

Index

*7, 19
.NET, 25
.class, 23, 163, 280
.java, 24, 280
.properties, 212

abstract, 130
accesor, 98
acoladă, 284
ActiveX, 11
Add/Remove Programs, 268
adresă, 64
agregare, 116
alocare de memorie, 80
alternativă, 354
ant, 275
antet, 99
apelare prin referință, 70, 88
API, 24, 33, 310
applet, 21, 29, 34
appletviewer, 30
argument, 86
arhivă, 103, 322
aritmetică pe adrese, 66
ArrayIndexOutOfBoundsException,
81, 188
ArrayList, 218
aruncare de excepții, 179
ascunderea informației, 92, 98
atomicitate, 92

atribut, 93
author, 283
autodecrementare, 43
autoexec.bat, 274, 321
autoincrementare, 43
AWT, 29, 34

beta, 273
biblioteci, 273
binding, 125
bit, 46
Blackdown, 274
Blackdown JDK, 24
blocaj circular, 259
blocare, 242
blocare circulară, 240
Boolean, 311
boolean, 38, 295
break, 53, 58
browser, 302
buffer, 243
BufferedReader, 208, 214, 312
BufferedWriter, 312
Byte, 311
byte, 37
bytecode, 23, 280

C++, 23
C#, 25
câmp, 66, 95
Calendar, 312

- call stack, 179
- cast, 43, 163, 174
- catch, 180, 187, 204, 293
- char, 38
- Character, 311
- charAt, 75
- cheie, 212, 342
- choice, 353
- ChoiceFormat, 313
- clasă, 37, 88, 91, 93, 110, 294
- clasă abstractă, 174
- clasă anonimă, 151, 156, 174
- clasă de împachetare, 142
- clasă de bază, 174
- clasă derivată, 117, 174
- clasă exterioară, 145
- clasă finală, 174
- clasă interioară, 115, 145, 174
- clasă locală, 151, 153, 174
- clasă wrapper, 174
- Class, 163, 174
- class, 282
- Class-Path, 330
- ClassCastException, 163
- CLASSPATH, 103, 110, 274, 277, 320
- client-side, 27
- clone, 70
- cod mașină, 22
- colecție de resurse, 212, 214, 335
- colecție implicită, 344
- colectare de gunoaie, 69, 88
- comentariu, 36, 281, 286
- compareTo, 74
- compareToIgnoreCase, 75
- compilator, 26, 273
- compresie, 322
- concatenare, 73
- consolă, 274
- constantă, 38, 100, 296
- Constructor, 167
- constructor, 96, 110, 120, 282
- constructor implicit, 121
- construire, 88
- consumator, 243
- context, 224, 228
- continue, 53, 58
- Control Panel, 268
- convenții de notație, 279, 294
- conversie, 43, 79
- conversie de tip, 66
- coordonare, 242, 245, 259
- countTokens, 209
- Created-By, 330
- daemon, 259
- Date, 101, 312
- date, 353
- DateFormat, 313, 338
- deadlock, 240
- declarare, 68, 80, 288
- deplasare, 46
- deprecated, 304
- dereferențierea, 66
- derivare, 117
- do, 51, 58, 292
- doclet, 301
- documentație, 299, 356
- Double, 311
- double, 38
- dynamic binding, 126
- early binding, 125
- Eclipse, 267
- editoare de text, 265
- EJB, 21
- else, 49, 291
- endsWith, 75
- eNews, 358
- enterprise, 11

- equals, 72, 74, 88, 96
- equalsIgnoreCase, 76
- eroare, 179
- err, 312
- Error, 181
- etichetă javadoc, 283, 303
- evaluare booleană, 45
- excepție, 67, 178
- excepție netratată, 182
- excepție runtime, 182, 205
- excepție tratată, 182, 205
- Exception, 181
- exception, 283
- excludere reciprocă, 229
- expandare inline, 143
- expansiune dinamică, 83
- expresie, 40
- extends, 119, 174

- fișier secvențial, 210
- fișier de proprietăți, 212, 214, 346
- fișier sursă, 22
- Field, 167
- File, 312
- FileInputStream, 312
- FileOutputStream, 313
- FileReader, 210, 214
- FileWriter, 194
- final, 100, 128
- finally, 191, 205, 293
- fir de execuție, 216, 259
- First Person, 20
- Float, 311
- float, 38
- flux, 40, 208
- for, 51, 58, 292
- fork, 217
- formatare, 338
- forName, 167
- Forte for Java, 267

- frame, 302
- friendly, 95, 110, 120
- ftp, 20
- funcție, 37, 95

- generic, 93, 115
- getBundle, 213
- getBytes, 76
- getClass, 164, 167
- getConstructors, 167
- getInt, 171
- getMethods, 170
- getter, 98
- glibc, 270
- Gosling, James, 19
- grup, 343
- GUI, 12

- Hashtable, 312
- hexa, 38
- Hoare, C.A.R., 229
- HotJava, 21
- HTML, 30, 299
- URLConnection, 313

- I/O, 207
- IBM, 22
- IBM JDK, 274
- identificator, 39, 58
- ieșire standard, 208
- ierarhie, 115
- if, 49, 58, 291
- implementare, 139
- implements, 174
- import, 102, 110, 281
- in, 312
- incapsulare, 92, 121
- inconsistență, 228, 259
- inconsistență la concurență, 228
- indentare, 283

- indexOf, 77
- indiciere, 79
- InetAddress, 313
- inițializare, 289
- inițializator static, 108
- inline, 143
- inner class, 145
- InputStreamReader, 208
- Instalare, 267
- instanțiere, 99
- instanță, 93
- instanceof, 67, 106, 108, 118, 164, 174
- instrucțiune, 44, 290
- instrucțiune simplă, 40
- instrucțiune vidă, 50
- int, 37
- Integer, 101, 311
- interfață, 115, 139, 174, 295
- interface, 139, 282
- interfețe multiple, 141
- internaționalizare, 335
- internet, 11
- interpretor, 28
- InterruptedException, 258
- intrare standard, 208
- intrerupere, fir de execuție, 228
- invariant, 117
- invoke, 170
- IOException, 182, 208
- isInstance, 164
- ISO, 337
- iterație, 44
- J2EE, 26
- J2ME, 26
- J2RE, 28, 33
- J2SDK, 28, 33
- J2SE, 26, 27, 33
- JAR, 274
- jar, 31, 322
- Java, 20
- java, 29, 33, 36
- Java 2, 27
- Java 2 API Specification, 310
- Java 2 Platform, 11
- Java 2 Platform Standard Edition, 277
- Java 2 SDK, 27
- java.io, 101, 311
- java.lang, 101, 311
- java.lang.Class, 238
- java.lang.reflect, 166
- java.net, 311
- java.text, 311
- java.util, 101, 311
- JavaBeans, 21
- javac, 29, 33, 36, 274
- javadoc, 278, 299
- JavaSoft, 21
- JavaWorld, 358
- JBuilder, 267
- JCreator, 267
- JDBC, 34
- JDK, 21, 27, 33
- jEdit, 266
- Jext, 266
- JFC, 21, 29
- jjikes, 26, 33, 274
- JNI, 34
- JRE, 277
- JSP, 21
- JVM, 23, 33
- Kaffe, 24
- kitul, 267
- lastIndexOf, 77
- late binding, 117, 126
- legare, 125

- legare dinamică, 117, 126
- legare statică, 125, 129
- legare târzie, 117, 126
- length, 75, 79, 88
- LIFO, 312
- linie de comandă, 86
- link, 305
- LinkedList, 312
- Linux, 267
- listă de discuții, 358
- ListResourceBundle, 312, 344
- Locale, 312, 337
- localizare, 337
- lock, 242
- Long, 311
- long, 38

- mailing list, 358
- main, 37, 59, 88, 99
- Main-Class, 330
- manifest, 329
- Manifest-Version, 330
- MANIFEST.MF, 328, 329
- Math, 101, 312
- matrice, 86
- membru, 95
- mesaj, 93
- mesaje dinamice, 351
- MessageFormat, 313
- META-INF, 328
- metaclasă, 163
- Method, 167
- metodă, 37, 56, 59, 92, 93, 282, 295
- metodă statică, 37, 59, 129
- Microsoft SDK for Java, 24
- moștenire, 93, 115, 174
- moștenire multiplă, 138
- modificator, 95, 98
- modificatori de acces, 282

- monitor, 229, 240, 259
- Mosaic, 20
- MS-DOS, 29
- MS-DOS Prompt, 36
- multi-dimensional, 86
- multiprocesor, 217, 224
- multitasking, 218
- multithreaded, 224, 230
- mutator, 98
- mutex, 229

- NEdit, 266
- Net Beans IDE, 267
- Netscape, 19
- new, 68, 88
- newInstance, 170
- nextToken, 209
- NoSuchMethodException, 182
- notificare, 243
- notify, 216, 242, 244
- notifyAll, 256
- null, 65, 88
- NullPointerException, 67, 68, 88, 182, 205
- number, 353
- NumberFormat, 313, 340
- NumberFormatException, 208

- Oak, 19
- obiect, 64, 67, 88, 91, 93, 110
- Object, 167
- octal, 38
- OOP, 18, 91
- operație, 92
- operator, 59, 71
- operator condițional, 56
- operatori, 40
- out, 312
- OutOfMemoryError, 181
- overloading, 58, 126

- overriding, 126
- overview.html, 302

- pachet, 101, 110, 294, 310
- pachet de resurse, 207
- package, 103, 110, 281
- package friendly, 105
- packages.html, 302
- pager, 11
- param, 283
- parametru, 37, 57, 68, 352
- parseDouble, 79
- parseInt, 79, 209
- partajare de resurse, 259
- PATH, 269
- PDA, 11, 27
- PDF, 301
- planificare, 224
- platformă, 28
- pointer, 23, 66, 88
- polimorfism, 93, 115, 117, 122, 175
- portabilitate, 322
- postfixat, 43
- prefixat, 43
- prerelease, 272
- println, 37
- printStackTrace, 189
- private, 94, 110, 120, 282
- procedură, 37
- proces, 217, 259
- producător, 243
- produs, 28
- programare generică, 141
- prompt, 29
- PropertyResourceBundle, 312, 344
- protected, 95, 120, 150, 282
- public, 94, 110, 120, 282

- race condition, 228, 259
- random, 81

- Reader, 104
- readLine, 208
- record, 93
- redefinire, 122, 126
- redefinire parțială, 128
- referință, 58, 64, 88, 116
- referință nulă, 65
- reflecție, 163, 165
- reflection, 165, 174
- relație, 116
- replace, 77
- resource bundle, 212, 342
- ResourceBundle, 312
- resurse Java, 356, 359
- return, 57, 59, 283, 291
- reutilizare, 92, 115
- reviste online, 358
- RMI, 34
- RPM, 270, 272
- RTF, 301
- RTTI, 116, 160
- run, 218, 221
- run-time binding, 126
- Runnable, 221
- RuntimeException, 182

- sablon, 142, 352
- SDK, 274
- secțiune critică, 229, 230
- secvență escape, 39
- see, 304
- semafor, 229
- semnătură, 58, 126
- serializare, 230, 259
- server-side, 27
- servlet, 21
- setInt, 172
- setPriority, 220, 243
- setter, 98
- shell, 271

- shiftare, 46
- Shockwave, 11
- Short, 311
- short, 38
- since, 305
- sincronizare, 229, 259
- sir, 79, 88
- sir de caractere, 39, 72
- sleep, 258
- Solaris, 267
- specializat, 117
- Stack, 312
- stare, 92
- start, 220
- startsWith, 78
- static, 99, 106, 282
- stivă de apel, 179
- stream, 208
- String, 312
- string, 72
- StringBuffer, 312
- StringTokenizer, 101, 209, 214
- structură, 92
- subclasă, 118, 175
- substring, 75
- Sun JDK, 274
- Sun Microsystems, 11, 17, 356
- super, 120, 174
- superclasă, 118, 175
- supraîncărcare, 58, 72, 126
- suprascrisoare, 126
- Swing, 29, 34
- switch, 54, 59, 292
- synchronized, 231, 233, 235
- System, 101, 312
- System.err, 208, 214
- System.in, 208, 214
- System.out, 208, 214
- target, 276
- 368
- telnet, 20
- template, 142
- The Duke, 19
- The Green Project, 19
- The Java Tutorial, 30
- this, 106, 108, 110
- Thread, 218, 312
- thread, 217, 218
- thread-safe, 218, 239
- ThreadGroup, 259
- throw, 179, 195, 205
- Throwable, 180
- throws, 197, 205
- time, 353
- tip primitiv, 37, 64, 71
- tip referință, 71
- toLowerCase, 78
- toString, 79, 96, 189
- toUpperCase, 78
- trim, 78
- try, 186, 205, 293
- tutoriale, 356, 357
- underscore, 39
- Unicode, 38
- unitate atomică, 92
- URL, 313
- User Variables, 321
- utilitar, 299
- valoare, 342
- variabilă referință, 64
- variabilă, 39, 296
- variabilă condițională, 244
- variant, 343
- Vector, 218, 312
- version, 305
- VisualAge for Java, 267
- vizibilitate, 95
- wait, 216, 242, 244

web, 11
WebRunner, 21
while, 51, 59, 292
Windows, 267
wrapper, 142

zip, 323
zonă de memorie, 67
zonă tampon, 243